TECHNISCHE UNIVERSITÄT DRESDEN

Energy-Aware Data Management on NUMA Architectures

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Ing. Thomas Kissinger
geboren am 25. April 1985 in Frankfurt (Oder)

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr. Peter Boncz
VU University Amsterdam
Faculty of Sciences, Department of Computer Science
Extraordinary Chair for Large Scale Analytical Data Management
1081 HV Amsterdam, Netherlands

Tag der Verteidigung: 23.03.2017

Dresden im März 2017

Abstract

The ever-increasing need for more computing and data processing power demands for a continuous and rapid growth of power-hungry data center capacities all over the world. As a first study in 2008 revealed, energy consumption of such data centers is becoming a critical problem, since their power consumption is about to double every 5 years. However, a recently (2016) released follow-up study points out that this threatening trend was dramatically throttled within the past years, due to the increased energy efficiency actions taken by data center operators. Furthermore, the authors of the study emphasize that making and keeping data centers energy-efficient is a continuous task, because more and more computing power is demanded from the same or an even lower energy budget, and that this threatening energy consumption trend will resume as soon as energy efficiency research efforts and its market adoption are reduced. An important class of applications running in data centers are data management systems, which are a fundamental component of nearly every application stack. While those systems were traditionally designed as disk-based databases that are optimized for keeping disk accesses as low a possible, modern state-of-the-art database systems are main memory-centric and store the entire data pool in the main memory, which replaces the disk as main bottleneck. To scale up such in-memory database systems, non-uniform memory access (NUMA) hardware architectures are employed that face a decreased bandwidth and an increased latency when accessing remote memory compared to the local memory.

In this thesis, we investigate energy awareness aspects of large scale-up NUMA systems in the context of in-memory data management systems. To do so, we pick up the idea of a fine-grained data-oriented architecture and improve the concept in a way that it keeps pace with increased absolute performance numbers of a pure in-memory DBMS and scales up on NUMA systems in the large scale. To achieve this goal, we design and build ERIS, the first scale-up in-memory data management system that is designed from scratch to implement a data-oriented architecture. With the help of the ERIS platform, we explore our novel core concept for energy awareness, which is *Energy Awareness by Adaptivity*. The concept describes that software and especially database systems have to quickly respond to environmental changes (i.e., workload changes) by adapting themselves to enter a state of low energy consumption. We present the hierarchically organized Energy-Control Loop (ECL), which is a reactive control loop and provides two concrete implementations of our Energy Awareness by Adaptivity concept, namely the hardware-centric Resource Adaptivity and the software-centric Storage Adaptivity. Finally, we will give an exhaustive evaluation regarding the scalability of ERIS as well as our adaptivity facilities.

Acknowledgements

First of all, I want to thank my advisor Wolfgang Lehner for giving me the opportunity to work in this wonderful atmosphere on this thesis, for supporting my ideas and visions, for believing in me, for (sometimes) respecting my non-standard wake-up times, and everything else...

I am grateful to Dirk Habich, Matthias Böhm, and Peter Boncz for co-advising my thesis as well as for the valuable discussions and hints. I want to thank my family for always supporting me throughout my way making all of this possible and also for proofreading the thesis. Thanks to my colleagues Benjamin, Tobias, Collie, Dirk (again), Timbo, Martin, Maik, Hannes, Ulrike, Julian, Tomas, Johannes, Patrick, Alex, Annett, Juliana, Lars (both), Robert, and everybody else.

I am thankful to all of my friends, who took care of the non-research part of my life following the "work hard, play hard" principle :) Especially Michi, Obschi, and Robi for the long fruitful nights of party, fun, and being fly with each other.

THANKS!1!!

Thomas Kissinger
12. Januar, 2017.

Contents

1 Introduction

"Available energy is the main object at stake
in the struggle for existence and the evolution of the world."
– Ludwig E. Boltzmann
(Philosopher and Physicist, 1844-1906)

The ever-increasing need for more computing and data processing power demands for a continuous and rapid growth of power-hungry data center capacities all over the world. As the first comprehensive analysis of U.S. data centers energy consumption in 2008 [27] concluded: The energy consumption of such data centers is becoming a critical problem, since their power consumption is about to double every 5 years. Moreover, the report stated that in 2006 about 61 billion kilowatt-hours (kWh) – equal to about $4.5 billion in electricity costs – were consumed by data centers only in the United States.

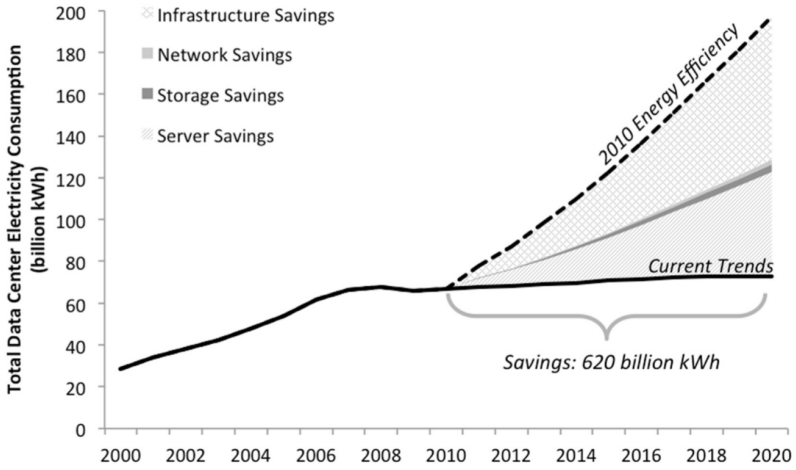


Figure 1.1: U.S. data center energy consumption: historic numbers, current trends, and 2010 energy efficiency scenarios [126].

However, a recently (2016) released follow-up study [126] revealed that this threatening trend was dramatically throttled within the past years, due to the increased energy efficiency actions taken by data center operators. As Figure 1.1 shows, the increase of U.S. data centers energy consumption only increased by 24 % from 2005 to 2010 and by 4 % from 2010 to 2014. The results of the study also emphasize that making and keeping data centers energy-efficient is a *continuous task*, because more

and more computing power is demanded from the same or an even lower energy budget, and that this threatening energy consumption trend will resume as soon as energy efficiency research efforts and its market adoption are reduced. For instance, Figure 1.1 additionally shows how the energy consumption would have been increased, if no additional energy efficiency improvements would have happened since 2010. Such a lack of innovation would have led to an estimated overall energy wasting of 620 billion kWh between 2010 and 2020.

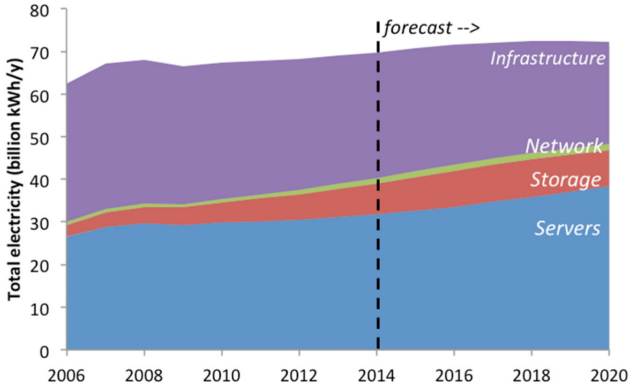


Figure 1.2: Historic, current, and predicted energy consumption by data center components in the United States [126].

The study also provides a breakdown of the overall energy consumption into the individual components of a data center. Figure 1.2 depicts their respective energy demand over time. As shown, the servers themselves are responsible for most of the energy draw and their overall share is predicted to increase in the future. Another high fraction of the overall energy consumption accounts for the infrastructure, which is responsible for getting the power to the servers and most importantly for getting the generated heat away from them. As the chart shows, the energy costs for infrastructure already decreased and are predicted to further decrease in future. The least share of energy is consumed by peripheral equipment such as network infrastructure and external storage systems. Thus, we can conclude that most of the energy saving potential arises from the servers themselves, because they are the most energy consuming components in a data center and every energy saving that is made within the servers also positively affects the absolute energy costs for infrastructure in terms of decreased power transformation losses and less energy consumed by cooling equipment.

To classify existing techniques that contributed to energy efficiency in data centers so far, we provide a classification in Figure 1.3. In this classification we distinguish three different research areas, which are the *infrastructure level*, the *inter-server level*, and the *intra-server level*. In the following, we will detail on the different topics:

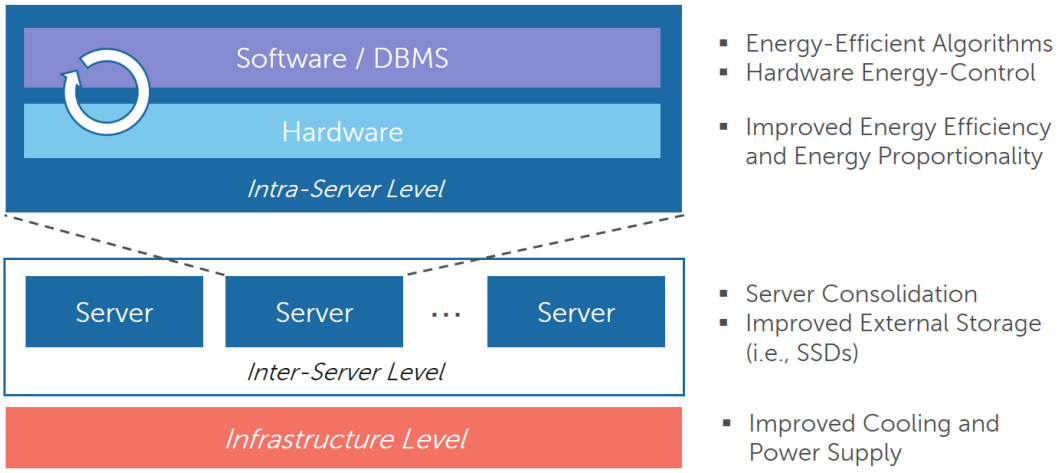


Figure 1.3: Classification of energy savings achieved in the past years.

Infrastructure Level. The main contributor for energy consumption at the infrastructure level is the cooling equipment that was designed to run at full speed independently of whether the full cooling power was actually needed or not. Today, this cooling equipment is designed as an on-demand system and provides much more sophisticated cooling strategies [103] such as airflow optimization [31], aisle isolation, economizers, and (warm-)liquid cooling [69].

Inter-Server Level. At the level of multiple servers, virtualization and server consolidation are the major complementary technologies that enabled the most energy savings [92, 136]. The basic idea is to break with the traditional paradigm of assigning a fixed set of application instances to a physical hardware resource. Instead, application instances are dynamically deployed at runtime to free hardware resources. Using this technology, servers are better utilized and thus, achieve a better energy efficiency compared to servers that are operated at a low utilization. For instance, its better to operate one server at 100 % than 10 servers at 10 %. Another major advancement in energy efficiency is the wide deployment of solid-state-drives (SSD), which provide better performance and consume less energy compared to mechanical hard drives [121].

Intra-Server Level. Energy efficiency advancements inside of a single server system mainly originate from the efforts of hardware manufacturers to build more energy-efficient hardware. Since higher core clocks became more and more infeasible because of their disproportional increase of power consumption, hardware vendors focus on alternative techniques such as multi-cores, superscalar pipelines, out-of-order execution, specialized instruction set architectures (ISA), and smaller feature sizes. Additionally, todays processors provide a rich set of energy-control features like dynamic voltage and frequency scaling (DVFS), core and processor sleep states, and HyperThreading. From

the software application itself, energy efficiency considerations are mostly limited to energy-efficient algorithms and operating system support for controlling the energy-control features of the processor. However, the main problem of leaving energy-control to the operating system only, is that it lacks important application-specific knowledge that is necessary for making appropriate decisions in terms of energy efficiency.

An important class of applications running in data centers are data management systems, which are an important component of nearly every application stack. While those systems were traditionally designed as disk-based databases that are optimized for keeping disk accesses as low a possible, modern state-of-the-art database systems are main memory-centric. Such in-memory DBMSs store the entire data pool in the main memory and use hard drives respectively SSDs only for durability purposes. Due to the high data volume, today's in-memory database systems have to manage, either scale-out or scale-up solutions are necessary to provide enough main memory capacities, because uniform memory access (UMA) architectures are strongly limited in their scalability. No matter whether a scale-up or a scale-out solution is employed, those architectures exhibit a non-uniform memory access (NUMA) that faces a decreased bandwidth and an increased latency when accessing remote memory, which needs to be taken into account when considering energy efficiency.

The fundamental difference between scale-up and scale-out approaches is the underlying programming model. While a scale-up system behaves like a single machine, database systems running on such an architecture are traditionally designed in a *transaction-oriented* way, because the scale-up architecture itself resolves memory accesses to remote memory transparently for the programmer using advanced cache coherency protocols. Contrary, scale-out systems require the programmer to do the remote memory access explicitly and thus, database systems on scale-out systems are commonly designed as *data-oriented* systems where all data objects are implicitly partitioned and each partition is placed in a well-known memory location. While *transaction-oriented* architectures let a transaction directly access data objects, which adds costs for latching data objects as well as for remote memory accesses, the *data-oriented* architecture needs to split the transaction into smaller pieces based on the actual partitioning and requires explicit communication between those transaction pieces to enable transactions that work on different partitions. The obvious advantage of such an architecture are less costs for latching data objects and especially the improved locality when operating on data, which positively affects energy efficiency, because less time and energy is needed for transaction execution.

While the smallest unit a data partition can be assigned to is a single server in the scale-up scenario, recent works proposed to push the programing paradigm of the *data-oriented* architecture even further and applying it to scale-up systems [104, 106]. The core idea is to use more fine-grained partitioning and make a single hardware thread the smallest unit a partition can be assigned to. Hence, data object latching efforts can be almost eliminated, because only one thread is able to access a specific partition, and the effective data access is guaranteed to happen in local main

memory. Using this approach, the authors were able to achieve performance and scalability improvements for disk-based DBMSs on single-socket as well as small-scale multi-socket architectures. However, the main problem of such a fine-grained *data-oriented* architecture is its sensitivity to workloads that do not fit to the current data partitioning which causes an odd utilization of the individual worker threads. To solve this issue, the partitioning needs to be adjusted on the fly, which comes at the cost of increased energy consumption.

In this thesis, we investigate energy awareness aspects of large scale-up NUMA systems in the context of in-memory data management systems. To do so, we pick up the idea of a fine-grained *data-oriented* architecture and improve the concept in a way that it keeps pace with increased absolute performance numbers of a pure in-memory DBMS and scales up on NUMA systems consisting of up to 64 sockets and a total of 768 hardware threads and beyond. To achieve this goal, we design and build the first scale-up in-memory data management system – namely *ERIS* – that is designed from scratch to implement a *data-oriented* architecture.

With the help of the *ERIS* platform, we explore our novel core concept for energy awareness, which is *Energy Awareness by Adaptivity*. The concept describes that software and especially database systems have to quickly respond to environmental changes (i.e., workload changes) by adapting themselves to enter a state of low energy consumption. We will show that the *data-oriented* architecture already provides a solid foundation for quick adaptations, but still misses important changes, which are covered by our *Living Partitions* architecture that understands individual data partitions as evolving objects that are not bound to a specific hardware thread anymore. Finally, we present the hierarchically organized *Energy-Control Loop*, which is an reactive control loop and provides two concrete implementations of our *Energy Awareness by Adaptivity* concept:

Resource Adaptivity. As already mentioned, today's hardware provides a multitude of energy-control knobs (e.g., DVFS, core and package sleep states, and HyperThreading). *Resource Adaptivity* is an holistic approach for managing these hardware energy-control knobs at runtime. The approach is based on an adaptive socket-local energy profile that is used to quickly reconfigure the hardware in case of workload changes. Doing so, the *Resource Adaptivity* is able find the most energy-efficient degree of parallelism as well as core and uncore frequency settings for the current workload type and system load.

Storage Adaptivity. An important factor for the query processing performance of a database system is the physical layout of the data that needs to fit to the current workload, which is a moving target. Thus, there is not the one-size-fits-all physical layout and *Storage Adaptivity* is responsible for continuously adapting the physical storage to increase query performance and thus, energy efficiency. Due to the *Living Partitions* architecture, *ERIS* is able to switch the physical data representation at runtime on a fine-grained level without inducing a severe negative impact on running queries.

1.1 Summary of Contributions

The contributions of this thesis can be summarized as follows:

(1) We investigate the opportunities for saving energy in main memory database systems. Based on these insights, we derive our core concept of *Energy Awareness by Adaptivity* and name *Resource Adaptivity*, *Storage Adaptivity*, and *Data Placement Adaptivity* as specific implementations of this concept. We analyze and formulate the requirements for an energy-aware data management system architecture.

(2) We improve the *data-oriented* architecture for scale-up database systems to be employable for pure in-memory DBMSs and to scale on large NUMA systems. We provide a comprehensive evaluation including measurements for a proof of concept implementation showing the overall feasibility of the approach as well as performance and scalability measurements for a real database system. We conducted the experiments on a scale-up NUMA system consisting of 64 sockets and up to 768 hardware threads using our in-memory DBMS *ERIS*.

(3) Based on our formulated requirements for an energy-aware data management system architecture, we advance the *data-oriented* architecture and present the *Living Partitions* architecture as the final result. The *Living Partitions* approach mainly decouples individual partitions from hardware threads and gives them the freedom to evolve and adapt at runtime. Moreover, we implemented the necessary architectural changes in *ERIS* and present the *Energy-Control Loop* as the hosting component for adaptations at runtime.

(4) We analyze the energy tuning knobs of a current mainstream server system and quantify the impact of the respective knobs on performance and power. We propose energy profiles and describe how they can efficiently be generated and maintained at runtime. With the help of such profiles, we show how different workload types affect the optimal compute resource configuration in terms of energy efficiency and performance. Furthermore, we present *Resource Adaptivity* as an holistic approach for adaptive energy-control on scale-up in-memory DBMSs. We give an exhaustive evaluation of the *Resource Adaptivity* implementation regarding all important aspects including an end-to-end evaluation using a real world load profile.

(5) We propose our approach of modular and flexible micro storage engines that operate within the individual *Living Partitions* and allow a fine-grained *Storage Adaptivity* at runtime. We describe the changes that are necessary for the query processing model to support such adaptations and their impact on the query processing performance including an end-to-end evaluation using highly variable workload scenarios.

1.2 Outline

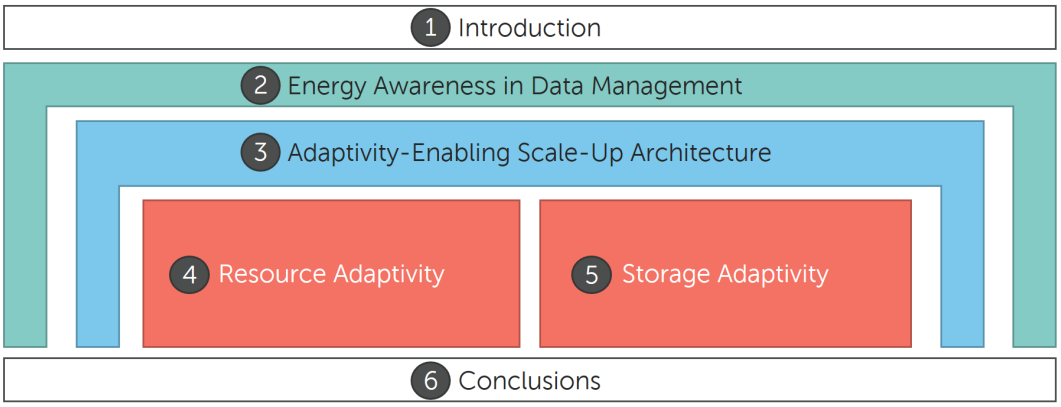


Figure 1.4: Structure of the thesis.

The structure of this thesis is visualized in Figure 1.4. Starting with Chapter 2, we present the necessary background for understanding the nature of energy especially in the context of data management systems. As the main outcome, we describe our core concept of *Energy Awareness by Adaptivity* and name possible implementations of this concept resulting in the formulation of a set of requirements for energy-aware data management systems. In the succeeding Chapter 3, we classify existing DBMS architectures with respect to our obtained requirements and argue that the *data-oriented* architecture is the best fitting foundation for our needs. We describe the enhancements that are necessary to transfer this architecture to a massively parallel in-memory database system and show first experimental results using a large scale-up NUMA system to demonstrate the feasibility of the approach. Based on our findings and requirements, we present the *Living Partitions* architecture, which enables the classical *data-oriented* architecture to do fine-grained energy-related adaptations at runtime and present a sophisticated experimental evaluation using our DBMS implementation *ERIS*. Afterwards, Chapter 4 and 5 deal with two implementations of our *Energy Awareness by Adaptivity* concept – namely *Resource Adaptivity* and *Storage Adaptivity* – each including a detailed evaluation. Finally, Chapter 6 concludes the thesis.

2 Energy Awareness in Data Management

In this chapter, we specify our target hardware architecture and cover the necessary background for understanding the nature of power and energy, especially in the context of data management systems. Furthermore, we introduce the topic of energy awareness in database systems and discuss the important aspects of how to measure and benchmark energy awareness in practice. As the main contribution of this chapter, we finally come up with our core concept of *Energy Awareness by Adaptivity* and formulate a variety of requirements that a database system architecture needs to fulfill to enable specific implementations of this concept. We derive those requirements based on three concrete implementations of our concept, which are *Resource Adaptivity*, *Storage Adaptivity*, and *Data Placement Adaptivity* as well as the overall concept itself.

2.1 Target Hardware Architecture

In this section, we specify the hardware architecture we are aiming at within this thesis. A schematic view of this architecture is given in Figure 2.1. As shown, we limit our scope to single-box scale-up symmetric multiprocessor systems and explicitly do not cover scale-out architectures. Symmetric multiprocessor architectures consist of multiple CPUs (also referred to as sockets, processors, or nodes) where each CPU nowadays implements multiple cores (multi-core processor) and simultaneous multithreading (SMT), e.g., Intel HyperThreading, to achieve parallelism within a single processor.

An important aspect of our target hardware architecture is the memory model. Here, we distinguish between a *uniform memory access (UMA)* model and a *non-uniform memory access (NUMA)* model. An UMA memory model is usually employed in single-processor systems due to its limited scalability, which is caused by a high contention on the memory bus and by increased latencies originating from an increased physical length of circuit paths when connecting multiple processors to the same memory. Thus, current multiprocessor systems employ a NUMA memory model where each processor has its own local main memory. To allow CPUs a transparent access to the remote main memory of another CPU, NUMA systems provide a globally shared memory by implementing a cache coherence protocol (e.g., MESI) to ensure consistent cache states across the individual processors. Database systems running on NUMA multiprocessing systems need to be aware of the physical data location as we will show in an in-depth analysis in chapter 3. The specific instances

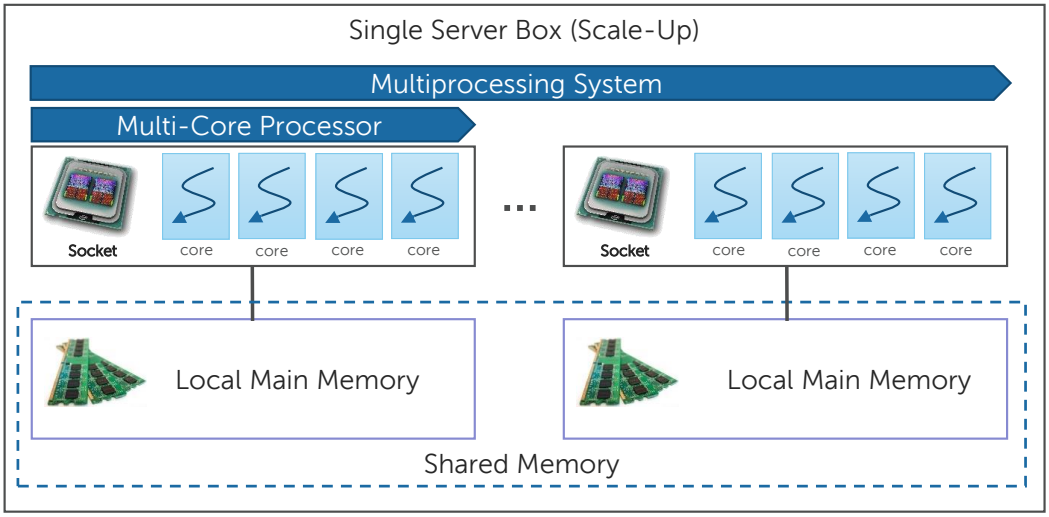


Figure 2.1: Schematic view of our targeted hardware architecture.

of our target hardware architecture range from widely deployed commodity server systems up to large scale-up systems (e.g., SGI UV familiy [124]).

To address the topic of energy consumption, modern processors offer energy-control mechanisms that are defined by the Advanced Configuration and Power Interface (ACPI) [137]. One mechanism defined by the standard are *processor states (C-States)* allowing the processor the put certain hardware components into a sleep mode. The standard includes about 10 C-States ranging from C0 to C7 (e.g., C1 and C1E) where C0 is the operating state and the C7 the deepest available sleep state. Such a processor state allows the CPU to selectively turn core clocks and caches on or off depending on their actual usage. Another mechanism defined by the ACPI standard are *performance states (P-States)*. P-states use the dynamic voltage and frequency scaling (DVFS) method to control the power consumption of the processor. The DVFS method enables the CPU to adjust it's frequency and supply voltage at runtime to control its power consumption and performance. While C-States and P-States are mostly standardized, additional energy-control knobs are available via machine-specific registers (MSR) that depend on the concrete processor model. We will give an in-depth overview and analysis of all available energy-control knobs of a current server system in chapter 4 and limit our considerations so far on performance and processor states.

To summarize, within this thesis we focus on large scale-up multiprocessing systems that employ a cache-coherent NUMA memory model. Such an architecture consists of multi-core processors that have a local main memory and provide certain energy-control features. The most important features for energy-control are processor states (C-States) to turn off entire hardware components and performance states (P-States) to adjust frequency and supply voltage at runtime.

2.2 Physical Definition of Power and Energy

The physical unit of electrical *Energy* (E) describes the amount of energy that was absorbed by an electrical circuit within a specific time span. Such an electrical circuit can be described using the basic model for discrete current depicted in Figure 2.2. The model consists of a linear power source, which mainly delivers energy including some energy losses modeled by an internal resistance (R_i), and the actual energy consumer (R_C) that converts the delivered energy to another type of energy such as thermal energy. The physical units that can be measured within such a circuit are the discrete *current* (I), the discrete *voltage* (U) and the *time* (t). To compute the electrical *Power* (P) that is drawn by the consumer at a specific point in time, formula 2.1 is used. As its unit shows, the power describes how much energy is consumed per second at a certain instant of time.

$$p(t) = u(t) \cdot i(t) = [V \cdot A] = [W] = [J \cdot s^{-1}] \tag{2.1}$$

To finally calculate the overall energy consumed, equation 2.2 needs to be used.

$$E = \int_t p(t) \cdot dt = \int_t u(t) \cdot i(t) \cdot dt = [W \cdot s] = [J] \tag{2.2}$$

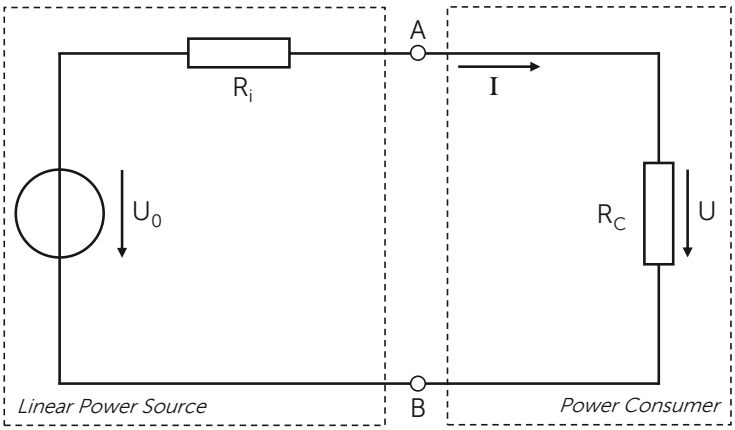


Figure 2.2: Physical model of a linear power source and a power consumer.

We schematically visualized the formula for electrical energy in Figure 2.3. As shown, energy is equal to the area under the power curve between certain time points, which makes it to a unit that can not be directly measured and needs to be approximated using methods of the numerical integration based on the measurements of voltage and current within the requested time range. One example for such an approximation is given by formula 2.3, which in practice requires $n + 1$ measurements of voltage and current at equidistant points in time. The quality of the approximation depends on the chosen value of n as well as the time span Δt.

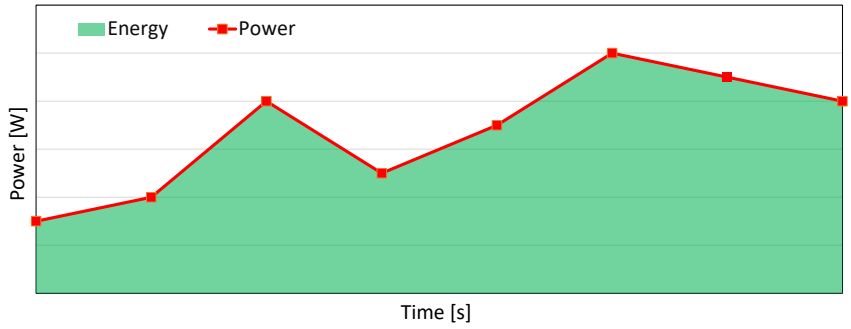


Figure 2.3: Energy as a function of the power curve.

$$E(\Delta t) \approx \overline{P} \cdot \Delta t \approx \frac{t_2 - t_1}{n} \cdot \left(\frac{p(t_1)}{2} + \sum_{k=1}^{n-1} \left(p\left(t_1 + k \cdot \frac{t_2 - t_1}{n} \right) \right) + \frac{p(t_2)}{2} \right) \quad (2.3)$$

Thus, we can summarize that electrical energy denotes the amount of energy absorbed by a consumer and is equal to the area enclosed by the power curve within a given time span, which needs to be approximated using frequent measurements of voltage and current.

2.3 Energy Awareness

In this section, we discuss certain metrics describing the energy awareness of a data management system. Beforehand, we need to define what energy awareness exactly is. In our understanding:

Definition 2.1 (Energy Awareness) *Energy Awareness is the ability of software (e.g., a database system) to be conscious of its energy consumption behavior related to the amount of work it is executing.*

According to the definition, there are two units involved in quantifying the energy awareness of software and thus, of a database system:

(1) The **Energy** consumption within a specific time span respectively the **Power** consumption at a specific point in time.

(2) The amount of **Work** executed by the database system within a certain time range respectively the **Performance** at a specific instant of time.

Hence, we will discuss in the following how the non-trivial unit of work respectively performance can be measured in a database system. Afterwards, we will introduce specific relations of work/performance and energy/power leading to the particular

metrics for energy awareness. These two metrics are *energy efficiency* and *energy proportionality* that we will discuss in detail especially in the context of the energy-control features available in modern processors. Both specific metrics for energy awareness are subject of further optimization to improve the overall energy awareness of a data management system.

2.3.1 Work and Performance Units

While energy and power are both units that are physically well defined and are measurable using hardware measuring devices, the quantification of work and performance is a more difficult process. As Figure 2.4 visualizes, we distinguish between five layers at which work and performance measurements can be taken starting from high-level application-specific layers ranging to low-level hardware-specific layers.

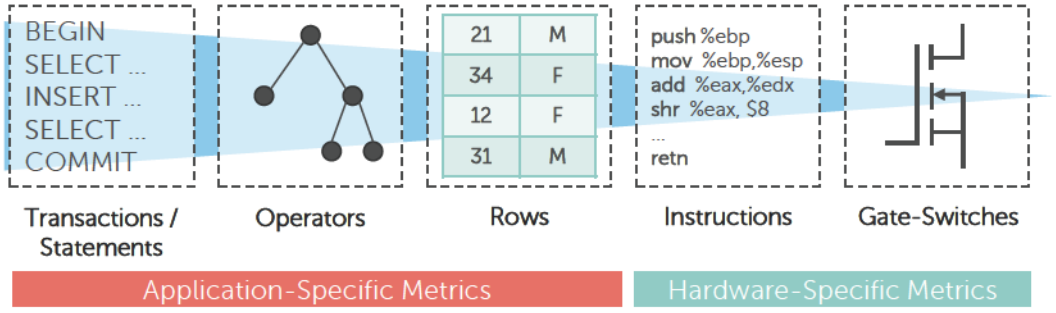


Figure 2.4: Available performance metrics in database systems.

Transaction Layer. In database systems, the highest available layer for measuring work is the transaction layer. On this layer, we can measure the amount of transactions that were executed using metrics like *number of transactions (#T)* for work or *transactions per second (Tps)* for performance. This performance metric is used by the TPC benchmarks [131] for energy efficiency evaluations and is applicable, because the queries are well-defined and always the same. An additional measurement that can be taken at this layer is the *transaction latency*, which is a valuable metric for quantifying the user experience respectively the quality of service (QoS). However, the transaction latency is not a suitable measurement for obtaining work or performance numbers, because it gives no information about the amount of work that was executed by the DBMS.

Operator Layer. In database systems, queries are usually compiled into query plans or data flows that consist of operators. Thus, the next available layer for doing work and performance measurements is the operator layer. Possible measurements on this layer are the *number of operators (#Ops)* executed for quantifying the work or *Operators per second (Ops/s)* for performance. Work

and performance measurements taken at those high-level layers face several problems. First, the amount of work a transaction or an operator executes highly depends on the queries that are executed. For instance, obtaining a performance measurement of 2 Tps or 10 Ops/s gives no information about the actual amount of work the DBMS was executing within that second, because neither the content of the transactions nor the executed operator is known. Another issue of high-level metrics is the measurement of performance, because transactions or operators are potentially running for a long time exceeding the measurement time span.

Row Layer. Relational database operators read, process, manipulate, and produce rows consisting of one ore more fields. Hence, the next lower layer for doing work and performance measurements is the row layer. Measurements that can be taken here are the *number of rows processed (#Rows)* for measuring work or *rows processed per second (Row/s)* for quantifying performance. Taking measurements for work and performance on this layer once again omits what an operator is actually doing while processing rows. However, measurements are taken at this layer or at operator layer, when calibrating work-energy models for energy-aware query optimizer [83, 143, 144].

Instruction Layer. The next available layer is the instruction layer, which is compared to the previous layers not application-specific anymore, because every application executes a series of instructions. Metrics that are obtained at this layer are *instructions retired (the number of instructions successfully completed)* for work or *instructions retired per second (Ips)* for performance measurements. Besides being application independent, work and performance measurements at this layer represent the real work that was actually executed by a hardware thread. An additional performance metric that can be derived at this layer is the *cycles per instruction (CPI)* value, which denominates how good the processor resources were actually utilized, because on modern superscalar CPUs instructions are executed simultaneously in different pipelines and its execution speed depends on a lot of factors (e.g., data locality and instruction dependencies). To actually measure the mentioned metrics, modern processors provide performance counters that can be used for efficiently obtaining those values [40, 91].

Gate-Switching Level. The lowest available layer for taking work and performance measurements is the gate-switching layer. Important metrics obtained at this layer are *the number if cycles (#Cycles)* performed by a chip for work or the *frequency (f)* for performance. This layer in combination with instruction layer is usually of interest for processor designers [25].

Hence, **we can conclude** that the choice of the appropriate work and performance measurement depends on the application area as well as measurement domain. While high-level application-specific work measurements are a suitable choice

for evaluating standardized database benchmarks covering the whole system, they are a bad choice for evaluating performance and work executed by single hardware threads or an individual processor as it can be done at the low-level application independent instruction layer.

2.3.2 Energy Efficiency

In this section, we introduce the first relation between energy and work respectively power and performance, which is called *Energy Efficiency (EE)* and is the most widely known relation of this kind. This metric is calculated as the quotient of work and energy when considering a certain time range or as the quotient of performance and power for a specific point in time as shown by Formula 2.4 (higher is better). For instance, database benchmarks such as the TPC benchmarks use the high-level *Watt per kilo-transactions per hour (W/KQph)* [131] unit, which is the inverse of the EE metric effectively resulting in the same relation except that lower is better in this case.

$$EE = \frac{\text{Work}}{\text{Energy}} = \frac{\Delta\text{Work}}{\Delta\text{Energy}} = \frac{\text{Performance}}{\text{Power}} = [W^{-1}] \tag{2.4}$$

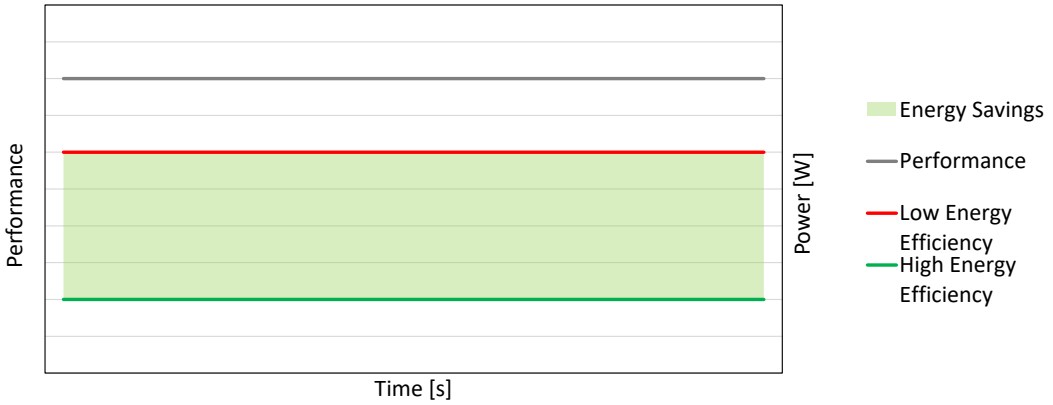


Figure 2.5: Schematic chart showing a low and high energy efficiency.

Thus, the overall goal for increasing the energy efficiency of a database system is to reduce the amount of energy for the same amount of work that was executed. For instance, if we assume a constant performance delivered by the DBMS as visualized in Figure 2.5, the amount of power drawn by the system defines the EE metric. Hence, a database system that consumes a high amount of power is less energy efficient compared to a DBMS that draws less power while delivering the same performance. This increase in energy efficiency can also be expressed by calculating the areas between the two power curves that is equal to the total saved energy.

To influence the power consumption behavior of a database system, modern processors provide different energy efficiency modes, which are known as performance

states (P-states) [137]. P-states use the dynamic voltage and frequency (DVFS) method to control the power consumption of the processor. The DVFS method enables the CPU to adjust it's frequency and voltage at runtime to control it's power consumption. However, the assumption of delivering a constant performance in different power modes does not hold anymore, because the performance reached by a processor – and thus, of the DBMS – actually linearly depends on the current frequency it is operating at as expressed by Formula 2.5. While there is a linear dependency between performance and frequency as well as voltage, the actual power dissipation of a CMOS technology chip depends on the frequency and the square of the supply voltage [98] (Formula 2.6). Additional influencing factors are the capacitance (C) and the active area (A) of the chip that is being switched.

$$\text{Performance} \propto f \propto V \tag{2.5}$$

$$\text{Power} \propto A \cdot C \cdot V^2 \cdot f \tag{2.6}$$

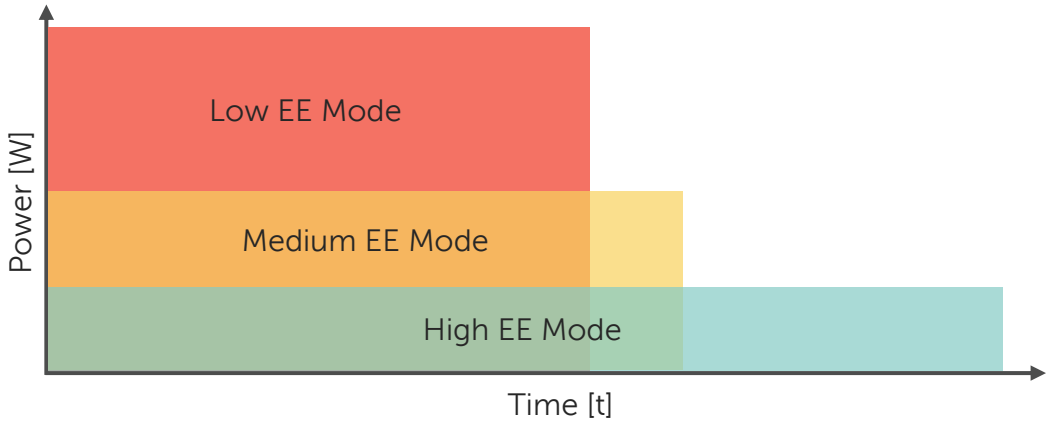


Figure 2.6: Impacts of different energy efficiency (EE) modes.

Due to the non-linear correlation between performance and power, the energy efficiency is different for varying performance states. Usually, a mode using a low frequency and thus, a low voltage results in a better energy efficiency according to Formula 2.5 and 2.6, while a high frequency and voltage results in the opposite effect. Hence, the logical choice is to use a low performance state to maximize the EE metric. Nevertheless, limiting the performance state scope to its respective minimum is not the solution for the problem, because a low performance state negatively affects the peak query execution performance as well as the query latency and thus, the user experience. For instance, as Figure 2.6 shows, the peak performance of the database system can commonly only be reached when applying the highest performance state that exhibits a low energy efficiency resulting in a low query latency. In contrast, operating in a low performance state mostly increases the energy efficiency, but results in higher query latencies.

To quantify this trade-off, an *energy-delay product (EDP)* metric is used that originates from power-performance trade-offs in integrated circuit designs [86, 16]. The general equation for the EDP is given in Formula 2.7. The EDP is the product of power (P) and latency (t), where the importance of latency is weighted using a weighting factor (W). A lower EDP value is preferred compared to a higher value.

$$\text{EDP} = P \cdot t^W \tag{2.7}$$

In practice, the weighting factors 1, 2, and 3 are used. A weighting factor of 1 results in the *power-delay product (PDP)* (Formula 2.8), which is equal to the area under the power curve (cf., example in Figure 2.6 and 2.3) and thus, to the energy. Substituting the weighting factor with 2, results in the basic *EDP* (Formula 2.9), which prefers latency and thus, performance over power using a squared impact of latency. Finally, using a weighting factor of 3 gives the *EDDP* or ED^2P (Formula 2.10), which is known as the voltage-independent version of the EDP due to the quadratic dependency of voltage an power (cf., Formula 2.6).

$$\text{PDP} = P \cdot t = E \tag{2.8}$$

$$\text{EDP} = \text{PDP} \cdot t = P \cdot t^2 \tag{2.9}$$

$$\text{EDDP} = \text{EDP} \cdot t = P \cdot t^3 \tag{2.10}$$

To summarize, energy efficiency is defined as the quotient of work and energy respectively performance and power. To improve the metric, modern processors use performance states (P-states) to adjust the trade-off between performance and power at runtime. Due to the non-linear correlation of performance and power as well as the existence of user-defined performance and latency demands, the appropriate choice a performance state is a non-trivial problem. Moreover, this problem becomes even more complicated when realizing that the performance of a DBMS depends on even more factors than just the processor frequency (e.g., memory clock, cache usage, and hardware contention) as we will show in Chapter 4.

2.3.3 Energy Proportionality

The second metric for energy awareness that is often forgotten or underestimated is *Energy Proportionality*. This metric denominates a proportional relationship between energy and work respectively power and performance as expressed by Formula 2.11. For instance, Figure 2.7 shows the performance demand of a database system over time, which depends on the number of active clients as well as the number and complexity of the queries that need to be processed. On the one hand, the chart shows the worst-case scenario in terms of energy proportionality, which is a constant maximum power draw that is completely independent from the requested

performance resulting in a high waste of energy. On the other hand, the chart includes the optimal power consumption behavior that proportionally adjusts itself to the actual performance needs.

$$(\text{Energy} \propto \text{Work}) \quad = \quad (\text{Power} \propto \text{Performance}) \tag{2.11}$$

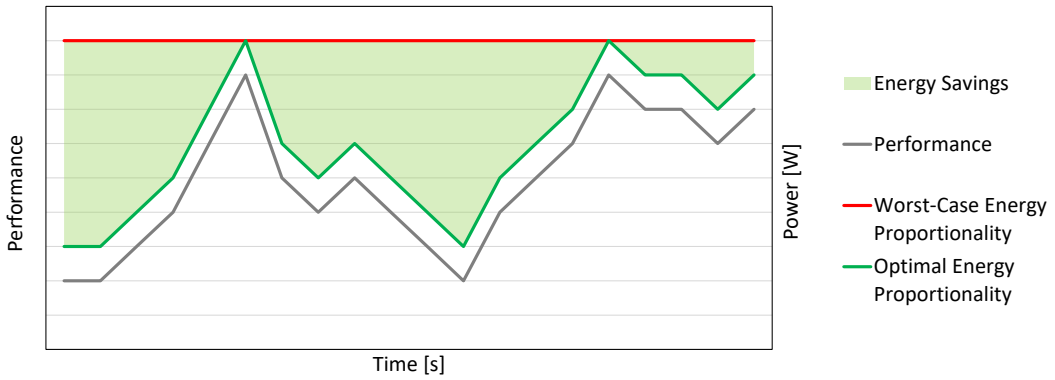


Figure 2.7: Worst-case and optimal energy proportionality.

The importance of a good energy proportionality arises from the fact that database servers are mostly fairly utilized. For instance, a study from 2007 [14] states that the typical utilization region of a server in operation is between 10 % and 50 %, because server capacities are usually over-provisioned to provide a reasonable performance in times of high load. The study also comes to the conclusion that the overall energy efficiency of a server can be doubled in real-life situations, if the energy proportionality is significantly improved.

Besides of the already mentioned performance states that allow an adjustment of voltage and frequency, the ACPI standard defines processor states (C-States) [137]. The standard includes about 10 C-States ranging from C0 to C7 (e.g., C1 and C1E) where C0 is the operating state and the C7 the deepest available sleep state. Such a processor state allows the CPU to selectively turn core clocks and caches on or off depending on their actual usage. The choice of the appropriate C-State is usually made by the processor itself and the operating system, which is able to give the processor hints based on scheduler information and is thus acting as an advisor. To make this issue tangible, Figure 2.8 shows a schematic example including the idle power consumption, the maximum possible power draw, the actual power consumption, and the energy efficiency as a function of the number of active physical cores of a modern processor for a fixed frequency respectively performance state. To calculate the energy efficiency, we assume an ideal constant performance increase for each core in operating state. As the example shows, the CPU is able to dynamically adjust its power consumption by putting idle cores into a deep sleep state.

However, the figure also shows that the *energy efficiency is not constant* as it would be optimal for energy proportionality. This non-linear behavior has two main

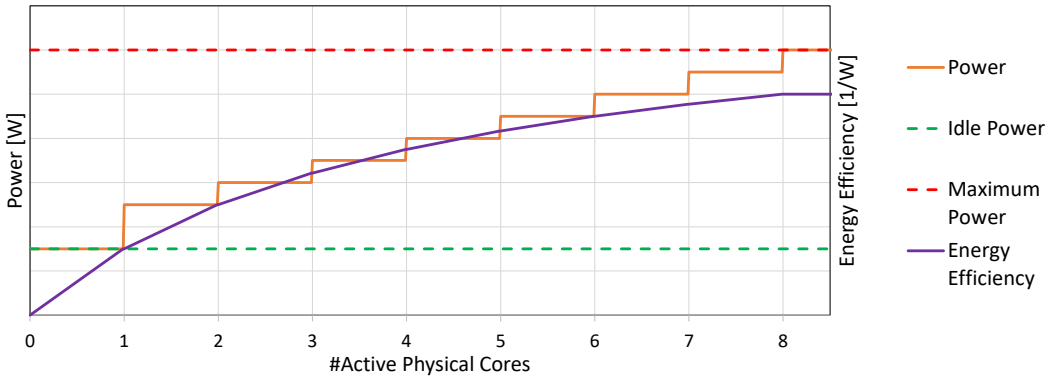


Figure 2.8: Schematic chart showing the processor power consumption depending on the number of active physical cores for a static frequency.

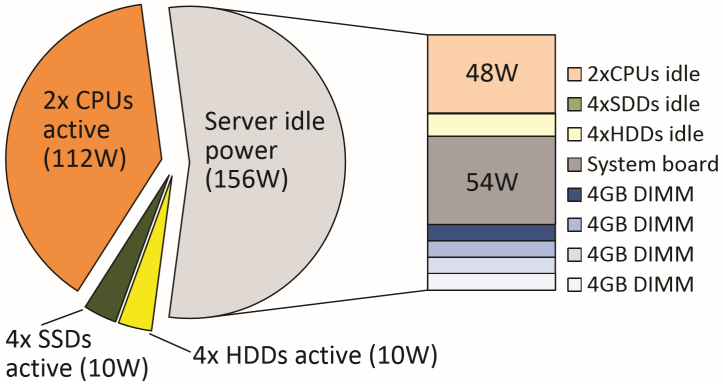


Figure 2.9: Power breakdown of a server in 2010 [134].

causes. First, the idle power consumption of a single server box amounts to a high fraction compared to the maximum power draw under full load, because of a static power consumption mainly originating from the power supply unit (PSU), the motherboard, and peripheral devices. Second, the high power costs for activating the first core that exist, because the CPU can enter its deepest sleep state only when all cores are in a sleep state allowing the processor to turn off the power-hungry last-level cache (LLC). Hence, the processor is only able to achieve it's peak energy efficiency, if all cores a fully utilized, but exhibits a bad energy efficiency in the typical operating range of a low to medium utilization. To put the issue of a high idle power consumption into numbers, Figure 2.9 shows the results of a measurement conducted in 2010 [134]. As the power breakdown shows, the server consumes more than 50 % of the peak power draw in idle mode, which is highly disadvantageous in terms of energy proportionality.

To cope with this issue, we classify existing works that contributed to the field of energy proportionality in *cluster-based* approaches and *single server* solutions, which is the subject of this thesis:

Cluster-Based. The available countermeasure in clusters consisting of multiple physical servers is the option to completely turn off entire servers with the overall goal to run only the necessary amount of servers at a high utilization. Doing so, the servers can operate in an energy efficient mode. When considering only a single server in isolation, this option of turning it on and off is not feasible, because of the high costs in terms of energy and time for booting and shutting down the system.

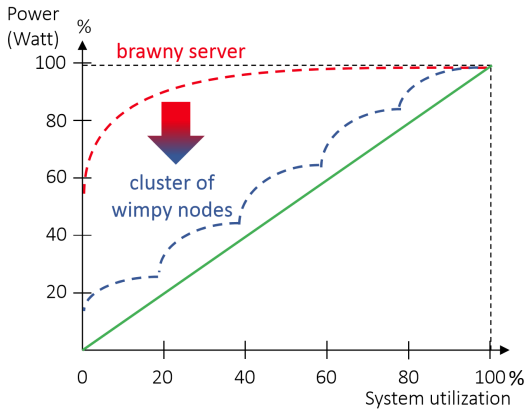


Figure 2.10: Energy proportionality of a brawny (large scale-up) server compared to a cluster of wimpy nodes [119].

Within cluster-based approaches for energy proportionality, we distinguish between *single-application scale-out* designs such as WattDB [119] and *multi-application* scenarios in cloud environments. The authors of WattDB propose that energy proportionality can not be achieved within a single server, because of the high static power costs (cf., Figure 2.9) and thus, energy proportionality optimizations are only feasible in clusters consisting of multiple servers or even wimpy nodes, which was also concluded by Tsirogiannis et al. [134]. As Figure 2.10 depicts, brawny scale-up systems are far away from optimal energy proportionality, while a cluster-based approach gets very close to the ideal curve. Nevertheless, this conclusion was valid in 2010, but does not hold anymore, because modern hardware significantly improved its energy proportionality as we will reveal in chaper 4. A similar approach is taken by works like GreenHDFS [71, 72, 85, 90] in the field of Hadoop clusters. Once again, the core idea is to concentrate the individual jobs on a small set of highly utilized servers to approximate energy proportionality in the large. However, similar to WattDB, those approaches need to shuffle data between the individual servers

to keep the entire data pool available, which faces additional energy costs and limits the agility of the approaches.

In cloud environments the principle for achieving energy proportionality is still the same, but the setting is a different one. The overall goal is to concentrate as many applications – not necessarily scale-out applications – as possible on a single server to increase its utilization. The main technologies exploited in this area are virtualization and the mentioned server consolidation [92, 136]. Individual applications are packaged into virtual machine images or lightweight containers that can dynamically be deployed on physical hardware resources based on the current server load.

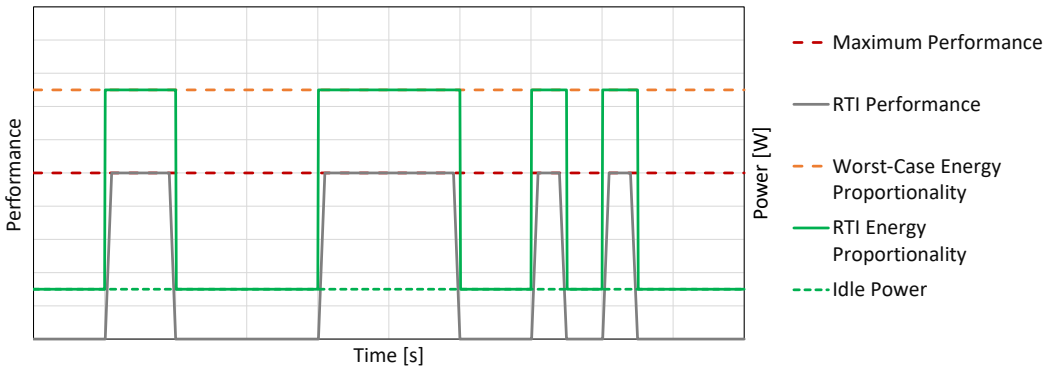


Figure 2.11: Schematic chart showing the worst-case and race-to-idle (RTI) energy proportionality.

Single Server. While turning entire servers on or off is a rather slow and energy consuming task in the scale-out approach, resource reconfigurations happen at the scale of microseconds within a single server system. Thus, a wide spread method for improving energy proportionality on a single server is the race-to-idle (RTI) method [14, 59]. The core concept of RTI is to leverage all available hardware resources to get a job done as fast as possible and return to idle mode afterwards until the next work package arrives. Thus, the server is either in idle mode or in the energy-efficient 100 % utilization mode. We provide an example for RTI-based execution is Figure 2.11. As the schematic example shows, every time a unit of work needs to be processed, performance as well as power consumption reach their respective maximum and the rest of the time is spent in idle mode, which still consumes the static power and gives a performance of zero. Nevertheless, since today's processors are highly parallel, the technique is only applicable either to highly parallel applications such as in-memory databases (intra-query parallelism) or applications that run a high number of requests in parallel (inter-query parallelism). Thus, state-of-the-art in-memory database systems implicitly implement the race-to-idle method

even if energy was not considered and can therefore be seen as a baseline for comparison.

Without the additional support of hardware features, like performance or processor states (P/C-States), software is limited in its opportunities for increasing energy proportionality. Since single servers respectively scale-up systems are the focus of this thesis, we will investigate the power management features of current processors in chapter 4 in detail and discuss relevant existing works.

To conclude this section, *Energy Proportionality* expresses a proportional relation between work and energy or power and performance and is a critical metric for reducing the energy footprint of a database system. To achieve energy proportionality, modern processors implement processor states to turn off unneeded cores. However, we also showed that energy efficiency depends on the utilization of a server and only reaches its respective peak for a high utilization while it is low in the typical operating range. Thus, the race-to-idle method is typically used to increase energy proportionality inside of a single server. However, while this method is mostly implicitly implemented, it does not make full use of the available power management features offered by modern processors.

2.4 Assessing Energy Awareness

So far, we introduced the units (e.g., work and energy) that are required to calculate the energy awareness metrics energy efficiency and energy proportionality. To provide a database system with the ability to assess its actual energy awareness for further optimization purposes, it needs to be provided with current measurements for the units of work or performance and energy or power. Thus, we will discuss in this section how those units can be monitored at runtime and how a benchmark for energy awareness must be designed to assess energy efficiency as well as energy proportionality.

2.4.1 Monitoring Energy Awareness

To enable software and especially database systems to reflect on their own energy awareness, applications must be capable of monitoring their own energy respectively power consumption as well as work or performance units to calculate derived metrics (i.e., energy efficiency and energy proportionality) that are subject of optimization. This monitoring information typically needs to be accessible for instance, to calibrate power-performance models or to enable feedback-driven approaches and finally to evaluate the overall energy savings achieved by a solution. Thus, we will detail in this section on how both units can be measured and monitored at runtime.

To monitor the energy respectively power draw of a server, hardware provides multiple measurement points that differ in their accuracy, resolution, domain, and accountability to specific applications. Figure 2.12 depicts an exemplary dual-socket

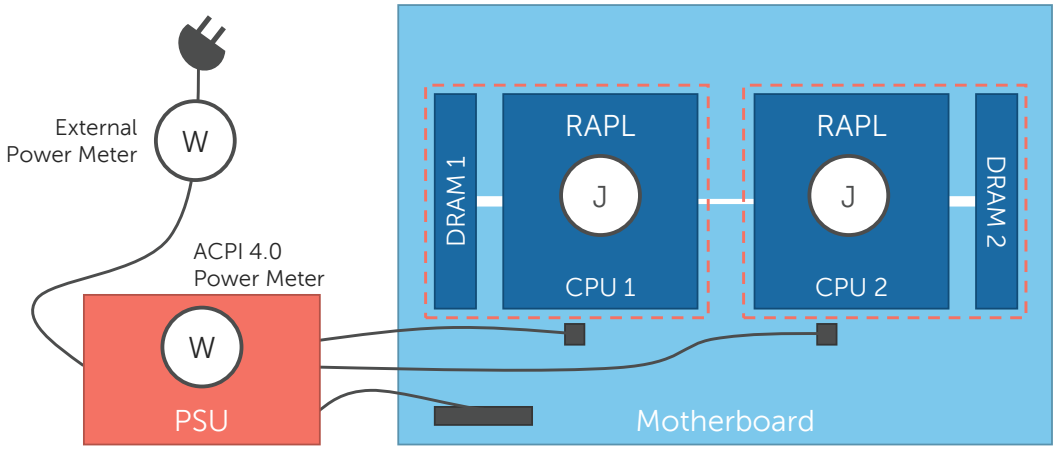


Figure 2.12: Available points for measuring energy or power in a dual-socket system excluding peripheral equipment (e.g., disks).

system equipped with a single power supply unit (PSU) and the available measurement point, which we will discuss in the following:

External Power Meter. The traditional way of measuring the power consumption of a server is to plug a power meter between the PSU and the electrical socket. Hence, an external power meter reports the numbers including all components of a server, which is equal to the effective power drawn by the entire system and should thus be preferred for an end-to-end evaluation. However, the usage of an external power meter faces multiple disadvantages. The most obvious drawback is that a power meter does not belong to the standard equipment of a server and needs an additional connection to communicate the measurements to the operating system. Moreover, accuracy and especially the resolution depend on the quality and therefore on the price of the power meter. For instance, a "Watts up?" power meter [129] ($100) provides an accuracy of 1.5 % at 1 Hz and a professional LMG450 power meter [147] (price on-demand) has an accuracy of 0.11 % at 20 kHz. Additionally, an external power meter gives only a coarse-grained view on the servers power consumption, which limits its applicability for calibrating energy models. Thus, we can conclude that the usage of an external power meter is not a practical approach that is limited in its benefit for energy optimizations, but should be used as a reference for end-to-end evaluations.

ACPI 4.0 Power Meter. The ACPI standard [137] defines a power meter for each power supply unit in server since version 4.0 to get rid of the need for an external power meter. An additional goal of the integrated power meter is to set power limits for server (power capping) to avoid power peaks that exceed the limits of the power supply infrastructures in a data center. However, the ACPI 4.0 power meter still faces the remaining drawbacks as the external power

meter solution does and as an additional obstacle, the ACPI power meter is mostly not implemented in current server systems and when it is implemented it mostly exhibits a low resolution.

Table 2.1: Available energy domains in RAPL.

Energy Domain	Affected Components
Package	The entire package/chip excluding the memory controller
DRAM	Memory controller and DRAM
Power Plane 0 (PP0)	All cores including L1/L2 caches
Power Plane 1 (PP1)	A special uncore device (e.g., integrated GPU)

RAPL. A more fine-grained solution for energy measurements inside of a server are *energy counters*, which are implemented per processor. For instance, in the dual-socket system shown in Figure 2.12, a set of energy counters is independently implemented by each of the two processors. Due to the near-total market dominance of Intel, RAPL counters [38, 68] are the most prominent implementation of such energy counters and are available since the Sandy Bridge generation. Energy counters are also implemented by AMD processors (APM) starting with the Bulldozer generation [9]. However, we will limit our scope to the RAPL implementation, because Intel processors take the major market share and provide the most sophisticated implementation of energy counters. RAPL stands for "running average power limit" and allows an application to obtain the energy consumption of certain CPU components (energy domain) between two specific points in time. Those energy values are accessible via machine-specific registers (MSRs) and are updated every millisecond. This high resolution allows an application to measure even short code paths [53]. Instead of using integrated power meters, processors use a calibrated energy model to estimate the energy consumption [117].

Table 2.1 enumerates all energy domains that are possibly available in a modern CPU. The actual availability depends on the processor model. For instance, a server-class CPU does not include the power plane 0 domain, because they do not have an integrated graphics unit, which is usually available in desktop-class processors. As another example, the Haswell-EP generation also does not include the power plane 0. RAPL counters allow an application to measure the energy consumption on a more fine-grained level compared to external or ACPI 4.0 power meters and additionally offer a much higher resolution. Moreover, RAPL counters are available in all modern Intel CPUs by default and thus, no additional costs incur. As the name already suggests, another purpose of RAPL counters is to define power limits (power capping) similar to the capabilities of ACPI 4.0 power meters. Nevertheless, this power limit is defined per processor and not per PSU.

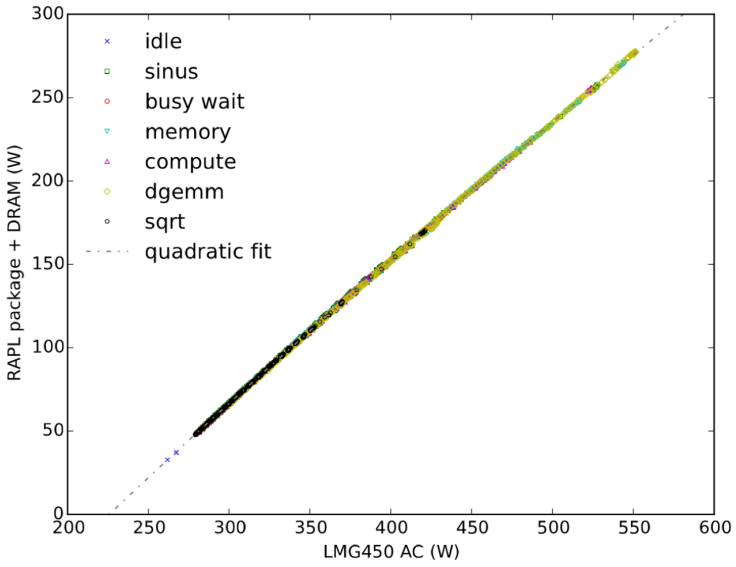


Figure 2.13: Comparison of power measurements with RAPL (package and DRAM) on Haswell-EP and total system power consumption using a high-accuracy power meter [50].

As already mentioned, RAPL energy counters use a calibrated internal energy model to estimate the actual energy consumption of the respective energy domain. Therefore, two questions arise. First, how accurate are those measurements? And second, how are those numbers correlated to the overall power consumption of the server (i.e., the measured by an external power meter). While the first question is hard to answer, because on-chip measurement comparisons are hard to accomplish, Hackenberg et al. answered the second question, which also gives a good hint for answering the accuracy question. The authors compared the RAPL energy measurements (package and DRAM domain) for a variety of different workloads to the numbers reported by a high precision external power meter (LMG450 [147]). It is worthwhile to note that the server was only equipped with a SSD as peripheral device and the fans operated at full speed. While the correlation accuracy was highly workload dependent for the Sandy Bridge EP generation [51], the newer Haswell-EP generation turned out to have an almost perfect correlation between RAPL and PSU energy respectively power measurements [50] as shown by Figure 2.13.

To summarize the opportunities for measuring energy and power at runtime, we can conclude that energy counters are the best way for obtaining appropriate results, because the most widely used RAPL implementation provides highly accurate measurements that correlate to the overall power consumption of a server and thus, the relative energy savings are highly accurate, too. Moreover, energy counters are in-built by default, allow fine-grained measurements at the level of single compo-

nents, and have a high resolution. However, to retrieve absolute values for the entire server, an external power meter needs to be used.

The other unit of interest that needs to be monitored at runtime is work respectively performance. As already discussed in Section 2.3, there are multiple ways of measuring those units depending on the actual purpose (cf., Figure 2.4). While application-specific metrics are usually trivial to obtain, hardware-specific metrics, e.g., instructions per second, instruction-per-cycle, or memory bandwidth, are hard or impossibly to obtain using software-based approaches such as static and dynamic code analysis, because the actual control flow and the time per instruction depends on so many factors (e.g., out-of-order, cache usage, and cache line contention). Thus, nearly all processor families implement hardware performance counters [68] (e.g., Intel, AMD, ARM, Sparc, and IBM POWER) today. The original intention of performance counters was to provide a better way to analyze and optimize the performance behavior of applications including database systems [127, 130]. Moreover, performance counters were initially used to estimate the power consumption of a processor [34, 146], which is now done via energy counters that also rely on the internal performance counters of the processor.

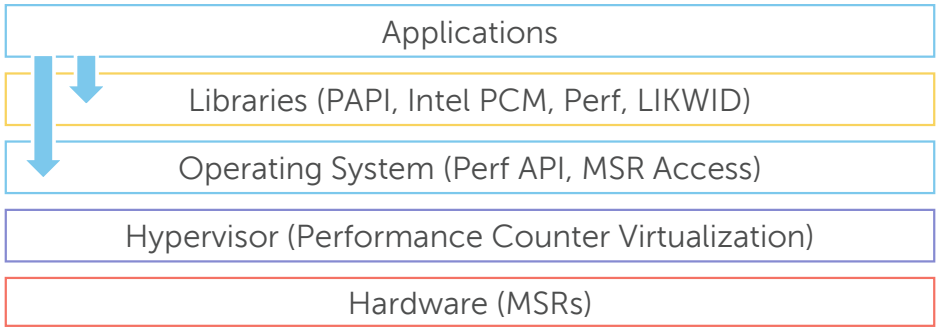


Figure 2.14: Performance counter access methods.

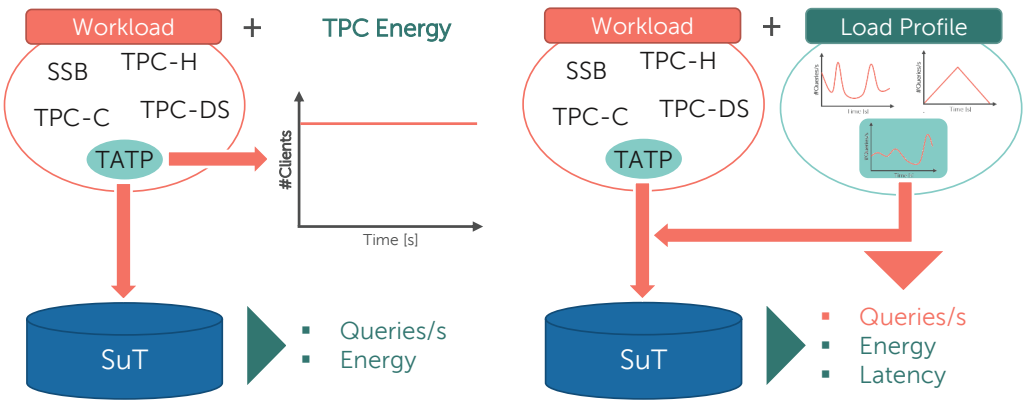
Since hardware performance counters are accessible by software, applications are able to monitor themselves. Today, performance counters are supported across the entire software stack ranging from performance counter virtualization [123] in virtualized environments over operating system support to specialized libraries as depicted in Figure 2.14. Hence, they are accessible by applications either by directly accessing the machine-specific registers (MSRs) or via libraries and kernel extensions [3, 28, 133, 141]. Both access methods have their respective advantages and disadvantages. For instance, while directly accessing the MSRs allows the access to all machine-specific counters, high-level APIs support features like multiplexing, due to the limited availability of programming slots, and the accounting to individual applications, which is only possible for unshared resources.

Modern processors offer a rich set of performance counters [68], which are either *hardware thread-specific* or *socket-specific*. Hardware thread-specific counters are simultaneously measurable for each hardware thread comprising counters for, e.g., instructions, core cycles, L1/L2 cache, branch events, and C-State residencies. Socket-specific counters are simultaneously measurable for each processor and include counters for, e.g., last-level caches, package C-States, uncore cycles, and main memory events.

Thus, **we can conclude** that hardware and software provide a sophisticated infrastructure for measuring hardware-specific work respectively performance metrics covering a wide range of component-specific measurements. However, since some components are shared across an entire processor, some measurements can not be accounted to a specific hardware thread that runs, for instance, a particular database operator. Contrary, high-level application-specific metrics such as transactions per second are trivial to implement in software and do not require additional hardware support.

2.4.2 Benchmarking Energy Awareness

Benchmarking is a critical step in the process of making data management systems energy aware, because energy savings need to be assessed in terms of energy efficiency and energy proportionality. The typical way of benchmarking database systems is to use a predefined benchmark that exactly describes how data is generated and which queries including the query parameters should be executed. Well known examples for such benchmarks are the TATP [62] and TPC-C [112] benchmark that simulate OLTP workloads and the TPC-H [132] as well as the star schema benchmark (SSB) [101] for OLAP workloads. The main metric that is evaluated by such benchmarks is commonly the amount of queries or transactions processed by the DBMS within a given time window (e.g., transactions per hour).



(a) TPC Energy specification.

(b) Advanced energy benchmark.

Figure 2.15: Energy benchmarking methods.

Due to the increasing relevance of energy concerns in data management, the Transaction Processing Performance Council (TPC) specified the *TPC Energy* extension [131] for the TPC benchmarks. This energy extension mainly describes how energy consumption should be measured during the benchmark and uses Watts per work per unit of time (e.g., W/kTpm) as primary metric for comparison. This metric is equal to the energy efficiency defined by Formula 2.4. The applicability of the TPC Energy extension is not limited to TPC-class benchmarks, due to its simplistic design. For instance, Figure 2.15(a) shows a possible setup for a TPC Energy benchmark. In the example, we use the TATP benchmark as foundation. This benchmark defines – the same as others do – that a constant (in terms of time) number of clients continuously executes the predefined queries so that the database system is fully utilized. After execution, the number of processed queries and the total energy consumption is returned. Both metrics can be used to calculate derived metrics like, average transactions per second and mean energy efficiency.

However, the value of TPC Energy results have a limited value in terms of energy awareness, because the assessment process completely omits energy proportionality and the fact that energy efficiency depends on the DBMS utilization. This observation was already criticized by Schall et al. [120], because the TPC Energy specification only includes energy measurements under full utilization and in idle mode. Thus, we propose to split benchmarks for energy awareness into the *workload specification* (Definition 2.2) and the *load profile specification* (Definition 2.3).

Definition 2.2 (Workload Specification) *The workload specification comprises the definition of the data generation process and related parameters like the scale factor and defines a set of queries executed against the system under test (SuT) including the query parameter specification.*

Definition 2.3 (Load Profile Specification) *The load profile specification defines the amount of transactions or queries per time unit (relative to the maximum achievable throughput) that are executed against the system under test (SuT) as a varying unit over time.*

Based on the workload and load profile specification, a valid benchmark for energy awareness consists of a particular workload and a load profile as visualized in Figure 2.15(b). Such a load profile is preferably obtained from server logs of real-world applications. The main difference compared to the TPC Energy extension is that not a constant number of clients is used to produce a maximum DBMS load. Instead, the load profile defines a varying number of queries over time that are executed against the database system. Using this approach, the benchmark considers energy proportionality and takes the dependency between energy efficiency and system load into account.

The metrics returned by such an energy awareness benchmark are the energy consumed and additionally the average query latency respectively the query latencies

of all processed queries. The number of queries or transactions executed metric is already defined by the load profile and is thus not considered as a result. Instead, the query latencies are of special focus for benchmark result comparison, which can be done in two ways. First, an instance of the energy-delay product (cf., Formula 2.7) can be used for comparison, because the database system can trade energy efficiency for query latency and the EDP quantifies this trade-off and gives priority to latency. The other way for interpreting the query latencies is to additionally specify a maximum query latency in the workload specification, which can be considered as a service-level agreement (SLA) as it is known in the real world. Hence, we can compare how often and heavy this query latency limit was violated respectively obeyed.

To conclude this section, we can state that existing benchmark solutions for energy awareness are very limited in their value, because extensions like *TPC Energy* measure energy efficiency of the database system only under full utilization and in idle mode. However, this benchmarking approach does not consider energy proportionality and the dependency between system load and energy efficiency. Thus, we propose to use a combination of a workload specification and a load profile specification to overcome those weaknesses. While the workload specification is already given by existing benchmarks, load profiles need to be standardized for valuable comparisons. A special focus of such a benchmarking approach for energy awareness are the query latencies, which can either be compared using the EDP or by quantifying violations against the SLA.

2.5 Energy Awareness by Adaptivity

"We see beautiful adaptations everywhere
and in every part of the organic world."
– Charles R. Darwin
(Naturalist and Geologist, 1809-1882)

Research and innovations are often inspired by the time-tested mechanics of nature as it is done in the area of bionics. One of the most essential and fundamental natural mechanics is evolution describing the concept that each life form is in a continuous struggle for negative entropy (free energy) that can only be won by adapting itself to environmental conditions and changes [37, 122]. As we observe in nature, organisms adapt themselves to get as much benefit as possible from the available energy (energy efficiency) and are able to frequently scale their energy expenditures based on the current performance demand (energy proportionality). A prominent example for those mechanisms is the human brain that is able to specialize its regions based on environmental conditions and adjusts its energy draw to the current performance needs [95].

Thus, we transfer the mechanics of evolution to software and database systems in particular. The resulting core concept – discussed in this thesis – is *Energy Awareness by Adaptivity*, which understands a DBMS as an organism that needs to

frequently adapt itself to cope with the critical issue of energy awareness. Moreover, the need for agile adaptivity facilities is not only limited to the energy topic. For instance, in our data-driven world modern applications call for database systems that can efficiently cope with a much richer variety and diversity of data beyond static schemata and static workloads [6]. Hence, adaptivity in database systems is required all over the place to keep pace with the demands of our ever-changing colorful world.

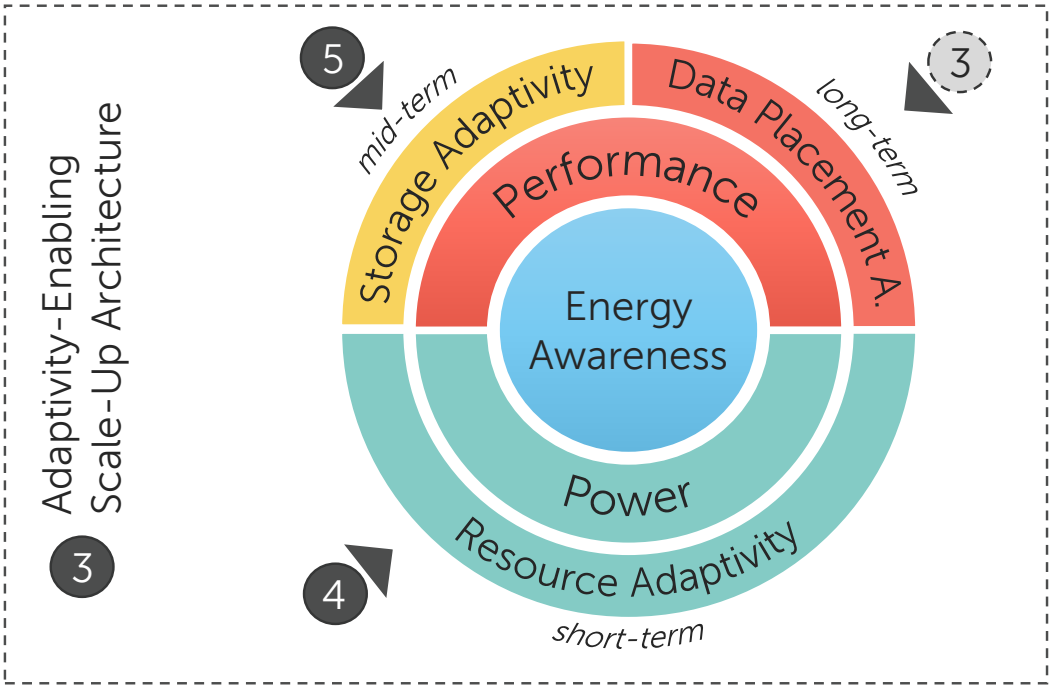


Figure 2.16: Energy Awareness by Adaptivity concept.

An important step for developing our concept of *Energy Awareness by Adaptivity* is to find feasible implementations to which we further refer as *Adaptivity Facilities* that do hardware-level or software-level adaptations to increase energy awareness. As a starting point, we have a look at the factors that actively influence energy awareness. Those factors (cf., Section 2.3) primarily are *power* consumption (lower is better) and *performance* (higher is better) as visualized in Figure 2.16. Hence, we will discuss possible adaptivity facilities that positively influence the individual factors including their adaptation time scale in the following:

Power. Decreasing the power consumption of a database system is mainly a matter of the underlying hardware. As already mentioned, modern hardware provides a rich set of energy-control knobs that can actively be controlled by software. Thus, we identify *Resource Adaptivity* as the first adaptivity facility that aims at actively doing hardware-level adaptations at runtime. Nevertheless, reconfiguring the hardware leads in certain situation to necessary software-level

adaptations. For instance, putting a core into a sleep state can force the database system to reduce its degree of parallelism. Since hardware reconfigurations happen at the scale of microseconds and the reconfiguration itself causes almost no additional energy costs, we consider resource adaptivity as a short-term measure to increase the energy awareness. Regarding the metrics for energy awareness, resource adaptivity is able to positively affect energy efficiency as well as energy proportionality (Table 2.2) as we will show in Chapter 4.

Table 2.2: Contributions of the individual adaptivity facilities to energy awareness.

Adaptivity Facility	Energy Efficiency	Energy Proportionality
Resource Adaptivity	✓	✓
Storage Adaptivity	✓	
Data Placement Adaptivity	✓	✓

Performance. Increasing the performance of database systems is a large and old research field ranging from query optimizers down to physical data layout optimizations. However, what we are looking for are fine-grained adaptation mechanisms that can frequently be invoked to increase the DBMS performance at runtime without the need for blocking entire data objects. While database systems are a software class that implements a lot of adaptivity features like different physical operators for a logical operation or different access path implementations, those adaptivity features are still too coarse-grained and limited in their agility for our requirements. Existing works that come close to our demands are, for instance, Micro Adaptivity [113] or Column Cracking approaches [65]. While the first approach is able to exchange the implementation of a physical operator during each operator call, the latter approach aims at a step-wise increase of the indexing granularity during query processing.

The first adaptivity facility we identified for increasing the performance of a database system is *Storage Adaptivity*, which aims at adapting the physical layout of data objects on a fine-grained level at runtime similar to the Column Cracking approach. However, this adaptivity facility is not solely limited to indexes. Instead, the goal of storage adaptivity is to cover physical layout optimizations in their entireness, with the overall goal of increasing the energy efficiency of the DBMS as shown in Table 2.2. Since physical storage adaptations induce additional costs in terms of energy consumption, this adaptivity facility is considered a mid-term measure. We will describe our approach for storage adaptivity including a discussion of the related works in chapter 5.

The second adaptivity facility addressing performance increases we want to outline is *Data Placement Adaptivity*. Since this thesis aims at large-scale NUMA systems, the physical location (i.e., on which socket) of data plays an important role. Thus, the physical placement of the individual data objects re-

spectively portions of them is crucial for the overall query performance. Using data placement adaptivity, the DBMS is able to increase the database systems energy efficiency as well as energy proportionality, because the appropriate placement depends on the current workload. Since physical data movement induces high costs in terms of energy, we consider data placement adaptivity as a long-term measure. Because concepts like data placement adaptivity are well investigated in database research [75, 104, 106, 111], we will limit the scope of this thesis to resource adaptivity and storage adaptivity except for a data placement adaptivity proof of concept on large-scale NUMA systems in chapter 3.

To summarize, we derived our core concept of *Energy Awareness by Adaptivity* inspired by the natural mechanism of evolution. Thus, we understand a database system as an organism in a continuous struggle for energy and hypothesize that frequent adaptations are the appropriate countermeasure to face this challenge. Based on our energy awareness considerations, we identified *Resource Adaptivity*, *Storage Adaptivity* and *Data Placement Adaptivity* as *Adaptivity Facilities* implementing our core concept.

2.5.1 Architectural Requirements

Implementing our core concept of *Energy Awareness by Adaptivity* and the corresponding *Adaptivity Facilities* requires a database system to fulfill a set of requirements, especially to allow adaptations on a fine-grained level at runtime without significantly affecting simultaneously running queries. Thus, we will use this section to derive those requirements (referred to as R-XX) for building an energy-aware DBMS architecture from our three adaptivity facilities as well as from general observations regarding energy.

To generally improve the energy efficiency metric of a DBMS, data object accesses need to be fast and thus, disk accesses should be avoided at all costs. Hence, only **main memory-centric (R-01)** architectures [128] should be considered when designing an energy-aware database system, since disk accesses slow down the query processing and thus negatively affect the performance metric. Regarding energy proportionality, a major requirement for the DBMS architecture is **scalability (R-02)**, because ideal energy proportionality can not be achieved when a database is not able to take an appropriate performance advantage from additional compute resources that consume additional energy. To achieve scalability, two extra requirements are implicitly necessary. First, **local data object accesses (R-03)** are required on large-scale architectures that usually implement a NUMA memory model to avoid costly remote accesses [107]. Second, **latch-free data object accesses (R-04)** are necessary for scalability, since locks and atomic instructions do not scale [70].

Resource Adaptivity Requirements. Resource adaptivity is mainly a matter of hardware reconfigurations. However, since reconfigurations at the hardware-level can cause single cores to be taken offline, the database architecture should

avoid explicit thread respectively operator to hardware mappings and thus, a **flexible work to hardware thread assignment (R-05)** is required. Moreover, methods like race-to-idle require **a high degree of parallelism (R-06)** to allow the DBMS to frequently turn entire processors on and off as well as keeping them highly utilized when being active.

Storage Adaptivity Requirements. To enable fine-grained storage layout adaptations at runtime, data objects need to be split into **small partitions (R-07)** to avoid a locking of the entire data object during the adaptation process. Furthermore, a **serialized partition access (R-08)** is necessary to allow the adaptation process to operate efficiently and to avoid additional latches inside of a partition. Since the physical storage layout is subject of change at possibly any point in time on a partition level, physical operators can not be bound at query compile time. Thus, a **late binding of physical operators (R-09)** is required where the actual binding can differ on a per partition level.

Data Placement Adaptivity Requirements. The workload of a database system is mostly a moving target. Thus, the internal partitioning of data objects should be adjusted to the current workload demands [106]. Hence, the DBMS architecture needs to support a **flexible partitioning scheme (R-10)** as well as **flexible partition to processor mapping (R-11)** on NUMA systems.

To summarize, a lot of requirements need to be fulfilled to build an energy-aware database system (summarized in Table 3.3). Those requirements either originate from general observations in terms of performance or from the individual *Adaptivity Facilities* of our *Energy Awareness by Adaptivity* concept. In Chapter 3, we will pick up these requirements and compare existing architectures for their ability to fulfill them. Additionally, we will propose the necessary extensions to the best fitting architecture.

2.5.2 Energy-Control Loop

To integrate our adaptivity facilities into the overall DBMS infrastructure, we use the *Energy-Control Loop (ECL)* as the hosting framework. The ECL follows the design principle of a closed reactive control loop [43] as depicted in Figure 2.17. Hence, the ECL is continuously monitoring the database system to respond to changes in the workload (cf., Definition 2.2) as well as in the system load (cf., Definition 2.3). Such monitoring information, for instance, comprises the current power draw and the query latency. The actual decision for an appropriate actuation (e.g., reduction of the degree of parallelism, frequency adjustments, or storage layout adaptations) is done by the controller, which knows how to steer the database system into a direction that may improves its target metrics. Because the ECL tries to optimize the energy awareness of the DBMS, power minimization and performance maximization are the main objectives (cf., Figure 2.16).

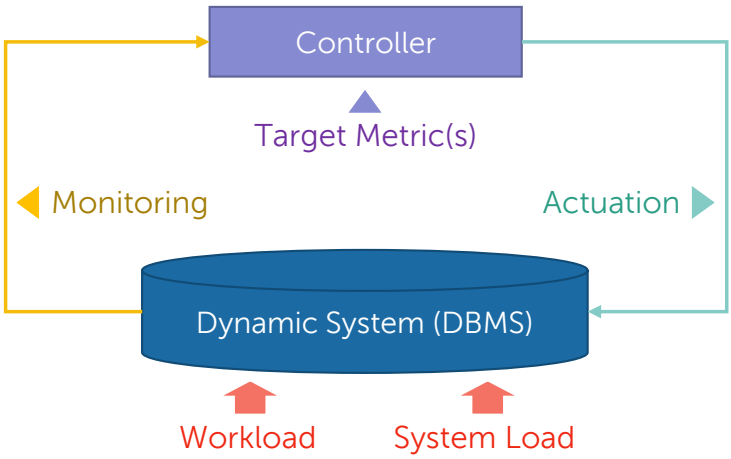


Figure 2.17: Block diagram of a closed reactive control loop in DBMS context.

The main issue of a closed reactive control loop is that it relies on periodic feedback (i.e., the monitoring data) that arrives within a static or dynamic interval Δt, which induces a delay $\leq \Delta t$ before the actual response to a change can happen. Moreover, the approach of a closed reactive control loop does not know the optimal response to a change, instead, it tries to steer the DBMS into a specific direction and in the next loop invocation the impact of the previous actuation can be assessed. To cope with the first issue (the delay), a look into the future is required to proactively prepare the database system for upcoming changes of influencing factors like the workload. This issue is typically addressed using forecasting techniques [26] that estimate the future based on the knowledge of the past. The second problem can be addressed by constructing a sophisticated model for the target metrics including a dependency of the influencing factors (i.e., workload and system load). However, for a complex dynamic system such as a DBMS, this model would depend on a vast amount of additional hardware-specific and DBMS-specific factors making it nearly impossible to construct and to maintain in practice. Thus, we will stick with self-learning coarse-grained models that are obtained using the feedback information of the database system as well as benchmarking-based approaches for the initial model construction.

As already discussed in Section 2.3, certain performance and power metrics are only available at a specific level of the software stack respectively for particular hardware components. Moreover, the implementations of our adaptivity facilities also address different software components. Thus, it is a natural decision to design the energy-control loop in a hierarchical fashion as visualized in Figure 2.18. The highest level in this hierarchy is the system-level ECL that exists once per DBMS instance. On this level, we are able to measure performance metrics of the transaction respectively query manager (cf., Figure 2.4), because such metrics are only available for the whole database system instance. For the resource adaptivity facility, the power consumption is the main focus and as outlined in Section 2.4.1,

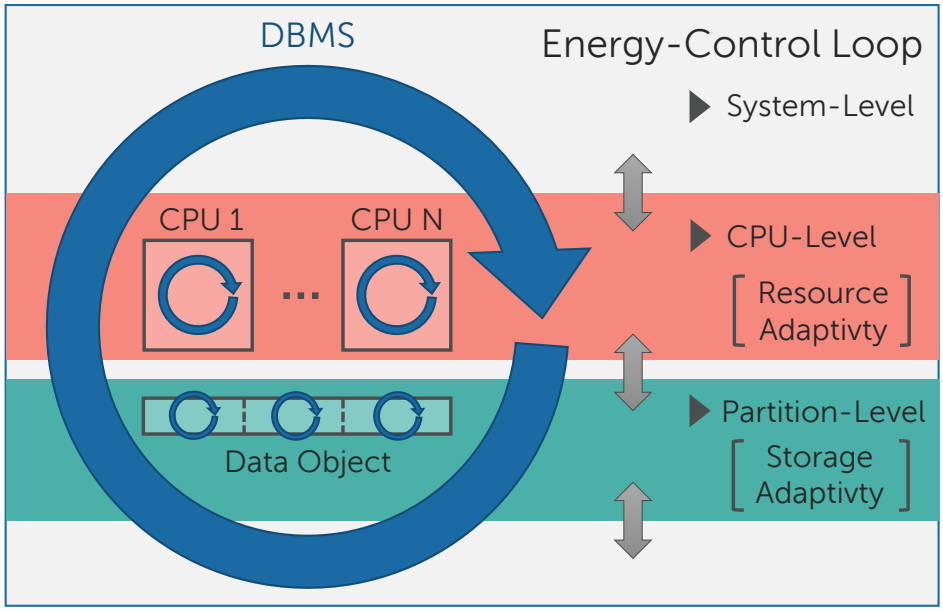


Figure 2.18: Hierarchical organization of our Energy-Control Loops (ECL).

RAPL energy counters use a single CPU as the smallest domain for power and energy measurements. Hence, resource adaptivity uses a specialized ECL instance per physical processor. Another available level from the perspective of a database system are either data objects or single data object partitions. At this level, the storage adaptivity facility operates resulting in an additional specialized ECL for each partition, because control decisions are local to this scope. Nevertheless, while all ECL instances and specializations operate at different level and time scales, they still need to communicate with each other. For instance, a resource adaptivity ECL needs information about the current query latency to avoid violations of the latency limit or a storage adaptivity ECL needs additional hardware resources for adapting a partitions storage format and the control of available hardware resources is up to the resource adaptivity ECL.

To summarize, we employ the design principle of a closed reactive control loop for our adaptivity facilities, which is the *Energy-Control Loop (ECL)*. The ECL continuously monitors certain database system metrics and responds to workload and system load changes using the measures of the respective adaptivity facility implementation. Moreover, the ECL is organized hierarchically to consider the scope of available DBMS metrics, of their time scale, and of the adaptation itself.

2.6 Summary and Conclusions

In this chapter, we started with the specification of target hardware architecture, which addresses large scale-up multiprocessing systems that employ a cache-coherent NUMA memory model and consists of multi-core processors that have a local main memory and provide certain energy-control features. The most important features for energy-control we identified are processor states (C-States) to turn off entire hardware components and performance states (P-States) to adjust frequency and supply voltage at runtime.

Subsequently, we discussed the physical unit of electrical energy denoting the amount of energy absorbed by a consumer that is equal to the area enclosed by the power curve within a given time span. To measure energy and power consumption at runtime, we concluded that energy counters are the best way for obtaining appropriate results, because the most widely used RAPL implementation provides highly accurate measurements that correlate to the overall power consumption of a server and thus, the relative energy savings are highly accurate, too. Moreover, energy counters are built-in by default, allow fine-grained measurements at the level of single components, and have a high resolution.

Afterwards, we introduced the notion of *energy awareness*, which expresses the relationship of work and energy respectively performance and power. Hence, we discussed possible metrics for determining the work respectively performance of a database system and concluded that the choice of the appropriate work and performance measurement depends on the application area. While high-level application-specific work measurements are a suitable choice for evaluating standardized application benchmarks (e.g., the TPC family), they are a bad choice for evaluating performance and work executed by hardware threads as it is done at the low-level application independent instruction layer. For such low-level performance measurements, we discussed performance counters that are implemented in mostly all modern processors and cover a wide range of component-specific measurements.

As the main metrics for energy awareness, we introduced *energy efficiency* and *energy proportionality*. Energy efficiency is defined as the quotient of work and energy respectively performance and power. To improve the metric, modern processors use performance states (P-states) to adjust the trade-off between performance and power at runtime. Due to the non-linear correlation of performance and power as well as the existence of user-defined performance and latency demands, the appropriate choice a performance state is a non-trivial problem. Hence, a widely used metric for expressing this trade-off is the energy-delay product (EDP). The second metric for energy awareness is energy proportionality, which expresses a proportional relation between work and energy or power and performance and is a critical metric for reducing the energy footprint of a database system. To achieve energy proportionality, modern processors implement processor states (C-States) to turn off unneeded cores. However, we also showed that energy efficiency depends on the utilization of a server and only reaches its respective peak for a high utilization while utilization is low in the typical operating range.

In the next logical step we discussed benchmarks for energy awareness and concluded that existing benchmark solutions for energy awareness are very limited in their value, because extensions like *TPC Energy* measure energy efficiency of the database system only under full utilization and in idle mode. Nevertheless, this benchmarking approach does not consider energy proportionality and the dependency between system load and energy efficiency. Thus, we proposed to use a combination of a workload specification and a load profile specification to overcome those weaknesses. While the workload specification is already given by existing benchmarks, load profiles need to be standardized for valuable comparisons. A special focus of such a benchmarking approach for energy awareness are the query latencies, which can either be compared using the EDP or by quantifying violations against the service-level agreement (SLA).

Based on our in-depth discussions of energy awareness, we derived our core concept of *Energy Awareness by Adaptivity*, which is inspired by the natural mechanism of evolution. Thus, we understand a database system as an organism in a continuous struggle for energy and frequent adaptations are the appropriate countermeasure to face this challenge. We identified *Resource Adaptivity*, *Storage Adaptivity* and *Data Placement Adaptivity* as *Adaptivity Facilities* implementing our core concept. Nevertheless, to enable a DBMS for those fine-grained adaptations at runtime, a lot of requirements need to be fulfilled. Those requirements either originate from general observations in terms of performance or from the individual adaptivity facilities of our energy awareness by adaptivity concept. In the following chapter, we will pick up these requirements and compare existing architectures for their ability to fulfill them. Additionally, we will propose the necessary extensions to the best fitting architecture.

Finally, we proposed the *Energy-Control Loop (ECL)* as the hosting vessel for our adaptivity facilities as we proposed within our collaborative research center "HAEC" [4]. The ECL employs the design principle of a closed reactive control loop. Hence, the ECL continuously monitors certain database system metrics and responds to workload and system load changes using the measures of the respective adaptivity facility. Moreover, the ECL is organized hierarchically to consider the scope of the specific adaptation facilities including the availability of DBMS metrics and the respective time scale of adaptation.

3 Adaptivity-Enabling Scale-Up Architecture

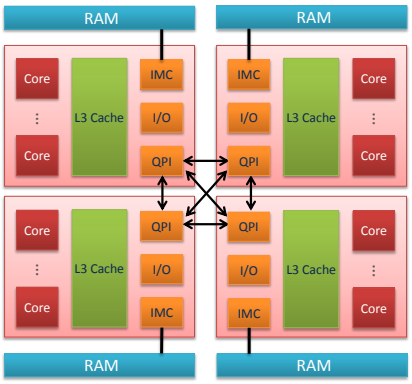
In this chapter[1], we pick up the requirements for an energy-aware database system that we derived in the previous chapter and propose – based on our requirements – the *Living Partitions* DBMS architecture, which is mainly designed for vertical scalability and adaptivity. Hence, we start with an exploration of current medium and large scale-up NUMA system architectures to especially assess the impact of remote main memory accesses. Based on those insights, we classify existing DBMS architectures in terms of their ability to scale-up on large NUMA systems as well as their ability to allow fine-grained adaptations at runtime. We conclude that the data-oriented architecture (DORA) provides us with the best foundation for fulfilling our requirements.

Nevertheless, this architecture still lacks (1) an investigation and appropriate concepts for in-memory DBMSs on large-scale NUMA systems as well as (2) certain requirements originating from our adaptivity facilities. To cope with the first issue, we will transfer existing concepts of the data-oriented architecture from medium-scale disk-based systems to large-scale in-memory systems, which is mainly a matter of the DORA-specific message passing subsystem that needs to keep pace with the increased speed of data object accesses. Moreover, we will give an in-depth evaluation of our prototypical implementation with regard to database primitives such as scans and index accesses. To address the second issue, we will extend the data-oriented architecture to enable adaptivity. Thus, we come up with the concept of *Living Partitions*. We will discuss the necessary changes to the previously discussed data-oriented architecture for large-scale in-memory database systems. Finally, we introduce our in-memory data management system *ERIS*, which was designed from scratch to implement the *Living Partitions* architecture as well as our *Adaptivity Facilities*. Furthermore, we will evaluate ERIS mainly in terms of scalability and compare the results to our previous proof of concept.

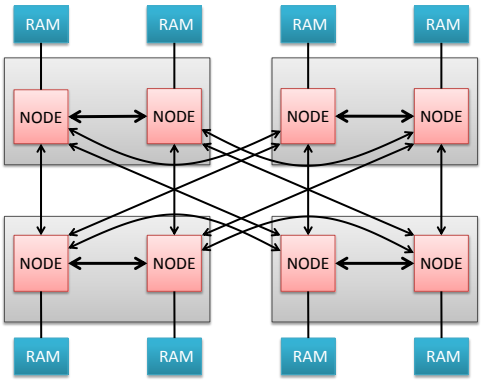
3.1 Scale-Up NUMA Architectures

As a consequence of the high main memory capacities in today's servers, modern database systems are very often in the position to store their entire data in main memory. Latency and bandwidth of the main memory are the major bottlenecks of such in-memory DBMSs. The significance of these bottlenecks increases when we

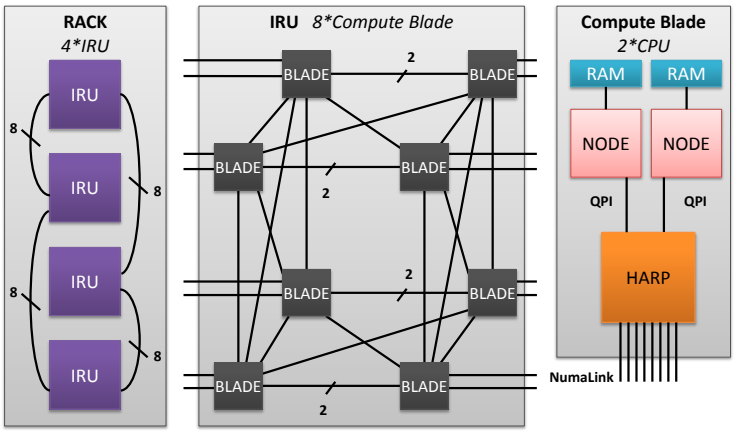
[1] Parts of this chapter are published in [78]



(a) 4-socket Intel machine (Detailed).



(b) 8-socket AMD machine (Topology View).



(c) SGI UV 2000/3000 (Topology View).

Figure 3.1: NUMA machines used for evaluation in the thesis.

consider the current trend towards tera-scale multiprocessor systems that exhibit a non-uniform memory access (NUMA). On NUMA platforms, each multiprocessor has its own local main memory that is accessible by other multiprocessors via a communication network. Database systems running on NUMA platforms face several issues such as the increased latency and the decreased bandwidth when accessing remote main memory. Additionally, we observe a worsening of the scalability of latches and atomic instructions. This is a result of the cache coherence maintenance overhead induced by the NUMA system. These issues are already measurable on wide-spread server systems consisting of four or eight multiprocessors. The demand for more parallel hardware forces vendors to put even more multiprocessors into a single server system (e.g., Oracle SPARC M6-32 [102] with up to 32 multiprocessors or the SGI UV family [124] that are sold as SAP HANA or Oracle In-Memory Machines). We also expect emerging technologies like 3D DRAM/CPU stacking to let NUMA characteristics appear in a single multiprocessor, where each core has

2-Socket Intel	4-Socket Intel	8-Socket AMD
2x Intel Xeon E5-2690 v3	4x Intel Xeon E7-4860	4x AMD Opteron 6274 (dual node)
24 cores (48 HW threads)	40 cores (80 HW threads)	64 cores
256 GB memory (128 GB per node)	128 GB memory (32 GB per node)	64 GB memory (8 GB per node)
30 MB LLC per sockets	24 MB LLC per sockets	12 MB LLC per socket
QPI: 19.2 GB/s per link	QPI: 12.8 GB/s per link	HyperTransport: 12.8 GB/s per link

SGI UV 2000	SGI UV 3000
64x Intel Xeon E5-4650L	64x Intel Xeon E5-4655 v3
256 cores (512 HW threads)	384 cores (768 HW threads)
8 TB memory (128 GB per node)	
20 MB LLC per socket	30 MB LLC per socket
QPI: 16 GB/s to HARP	
NumaLink6: 2x 6.7 GB/s between HARPs	

Table 3.1: Specification of evaluation machines used in this thesis.

its local low latency and high bandwidth main memory [77, 94]. Moreover, current Intel processors starting with the Haswell-EP generation already exhibit NUMA characteristics within a single processor. To allow database systems to scale-up on today's and future platforms, NUMA awareness has to be considered as a major design principle for the fundamental architecture of a database system.

In the following, we introduce NUMA system architectures and present low level benchmark results of the NUMA machines used in our experimental setup. As a reference throughout the section, the architectures of these systems are depicted in Figure 3.1. As bottom line of this section, we will derive fundamental design principles for NUMA-aware database system architectures.

3.1.1 NUMA System Architecture

NUMA systems consist of several interconnected multiprocessors, that are also referred to as *nodes*, *processors*, *sockets*, or *CPUs*. Each multiprocessor contains multiple processing units (cores) and an integrated memory controller (IMC). Consequently, the installed main memory is distributed among the IMCs in different

multiprocessors. However, each multiprocessor can access each memory location. Thus, latency and bandwidth of memory accesses depend on the distance between the requesting multiprocessor (source node) and the multiprocessor that contains the data (home node). The *local memory* associated with each multiprocessor is accessed with low latency at a high bandwidth. In contrast, *remote memory* is accessed via point-to-point connections [60, 66] between the multiprocessors that add latency and limit the achievable bandwidth. In the worst of our cases the latency of remote access is approximately 10 times higher and the bandwidth is limited to about 11% in comparison to local accesses.

Multiple levels of caches are commonly used to mitigate the performance impact of the above-mentioned latency and bandwidth constraints. The caches are distributed over the multiprocessors as well. All currently available NUMA systems enforce cache coherence to maintain a consistent view of all processing units on the shared address space. Small-scale NUMA systems with a manageable amount of nodes typically rely on snooping based cache coherence protocols that involve frequent broadcasts of requests to all multiprocessors. It has been shown in prior work [52] that the overhead of the coherence protocol caused by accesses to shared data can be very severe in such systems. In contrast, larger systems like the SGI UV 2000 or 3000 usually implement directory based cache coherence protocols between the nodes. SGI, e.g., uses NumaLink to connect blades with one another while the two nodes in each blade are connected to a hub via their Intel QPI links. The hub presents itself to the nodes as an external memory controller that participates in the snooping based coherence protocol. However, the requests are not broad-casted to all the hubs in the system. Instead, requests are only forwarded if the corresponding directory entry indicates a remote copy.

Naturally, data placement is an important aspect to consider with NUMA systems and data should be located close to the multiprocessor that accesses it frequently. The default data placement policy of Linux is called *first touch*. Newly allocated memory is placed local to the thread that actually writes (touches) it for the first time. It is, however, possible that memory is allocated on remote memories. Moreover, the default thread scheduler in Linux operating systems may migrate threads frequently to different multiprocessors, although it prefers intra-node thread migrations to inter-node migrations. This leads to remote memory accesses, even when the memory was allocated locally in the first place. Hence, the operating system leaves many opportunities for suboptimal (i.e., remote) memory access patterns. This is especially true, when many threads access a large portion of the main memory.

4-Socket Intel Machine			**8-Socket AMD Machine**			**SGI UV 2000/3000**		
distance	bandwidth (GB/s)	latency (ns)	distance (link width)	bandwidth (GB/s)	latency (ns)	distance	bandwidth (GB/s)	latency (ns)
local	26.7	129	local	16.4	85	local	36.2/58.0	81
1 hop QPI	10.7	193	1 hop HT (full link)	5.8	136	2nd processor	9.5/11.9	400
			1 hop HT (split,single)	4.2	152	1 hop NUMALink	7.5	505 - 515
			1 hop HT (split,dual)	2.9	152	2 hop NUMALink	7.5	625 - 635
			2 hop HT (split,single)	3.7	196	3 hop NUMALink	7.1	745 - 755
			2 hop HT (split,dual)	1.8	196	4 hop NUMALink	6.5	870

Table 3.2: Memory Read Bandwidth in GB/s and Read Latency in ns. Bandwidths are measured with concurrent sequential reads from all cores of the multiprocessor in order to maximize the amount of outstanding requests. Latencies are measured with a single thread that performs a pointer-chasing routine on memory allocated at different multiprocessors.

3.1.2 Low-Level Benchmark Results

Within this thesis, we use five different NUMA systems ranging from 2-socket machines to 64-socket machines with a total of 8 TBs of main memory. In Table 3.1, we summarized the hardware specifications of all machines. For our NUMA-related experiments, we use the *4-socket Intel machine*, the *8-socket AMD machine*, and the *64-socket SGI UV systems*. For our energy-related evaluations, we use the *2-socket Intel machine*, because the other systems either lack the implementation of RAPL counters or administrative issues deny us the access to them. To gain deeper insights in the performance of the four NUMA machines, we conducted several low level benchmarks. The best-case bandwidth and latency performances are an upper bound for the achievable performance and will help us to reason about the performance of our own algorithms. All measurements are performed with the BenchIT tool [52]. The results are shown in Table 3.2.

4-Socket Intel Machine. The 4-socket Intel machine with 4 multiprocessors is the smallest system we are considering for our NUMA experiments. The nodes of the Intel machine are fully connected via QPI links [66] as depicted in Figure 3.1(a). The results of our experiments show that the latency of remote memory accesses is only 50% higher than for local accesses. The impact of the QPI link on the achievable bandwidth is more severe as it results in 2.5 times lower data rates compared to local memory. However, the effects of the non-uniform memory access are small compared to the other two machines as communication between any two multiprocessors requires only one hop via QPI.

8-Socket AMD Machine. The second machine in our setup is an AMD machine. As shown in Figure 3.1(b), it is actually a 4-socket system where each socket houses a *dual node package*. The two nodes in a package communicate via HyperTransport [60], which practically results in a system with 8 multiprocessors. Each multiprocessor has four HyperTransport ports to connect to either the I/O subsystem or to other multiprocessors. As a unique feature of the AMD machine, HyperTransport links can be split into sublinks to connect a node with two other nodes with just one HyperTransport link. However, this results in different link bandwidths for different links. Additionally, even with split links, the AMD machine is not fully connected and certain routes require two hops.

As indicated in Figure 3.1(b), the two nodes that share a socket are connected via a dedicated (not split) HyperTransport link and can therefore utilize the full 16 bit link widths. Connections between other nodes are realized with 8 bit sublinks and hence have a lower connection bandwidth. Furthermore, some of the split links only have one sublink populated (denoted by *split,single* in Table 3.2) while both sublinks are occupied on other links (denoted by *split,dual*). Our experiments mirror these characteristics; depending on the distance of memory and accessing thread, we measure six different bandwidths and four

different latencies. The disparities between local access and the furthest remote access are a factor of 9.1 in bandwidth and 2.3 in latency.

64-Socket SGI Machines. The third and fourth machines in our setup is are a SGI UV 2000 respectively 3000 with 64 multiprocessors and a total of 8 TBs main memory. An overview of the topology is shown in Figure 3.1(c). Our system consists of 1 rack that houses 4 Individual Rack Units (IRUs). Each IRU consists of 8 Compute Blades, that in turn contain 2 multiprocessors each. Each socket is equipped with an Intel Xeon CPU with 128 GBs of local main memory. Both machines differ only in the CPU model, but share the same topology and communication network.

The two multiprocessors in a Compute Blade are connected via QPI to a communication hub called HARP. The HARPs are NumaLink hubs that connect the multiprocessors in a Compute Blade to other Compute Blades in the same as well as in other IRUs. As shown in Figure 3.1(c), each blade in our system has 8 connections to other blades. Each connection consists of two NUMA-Link6 links, one for each multiprocessor in the blade. The 8 blades in an IRU are connected as a 3D enhanced hypercube [125]. Each blade in an IRU is additionally connected to two blades in other IRUs. This topology leads to connections with up to four hops and six different bandwidths.

Measuring all possible distances reveals that the differences in bandwidth and latency between local access and the furthest remote access are as high as factor 5.5 and 10.7, respectively.

3.1.3 Design Principles for NUMA-Aware Database Systems

From the general NUMA architecture as well as our benchmark results, we derive that NUMA systems should be treated like a distributed system and that a scalable in-memory database system must be designed to maximize local memory accesses. Reading from or writing to remote memory suffers from up to ten times higher latencies and significantly lower bandwidths, hence remote accesses should be avoided whenever possible and batching should be considered for inevitable accesses to hide the bad latency. Furthermore, remote and concurrent memory accesses lead to cache concurrency as well as worse cache locality and hence, higher cache coherence overhead. However, small amounts of data that are mostly read from different sockets are unproblematic, because caches are able to hide the remote memory accesses.

As a conclusion, a scalable DBMS architecture for NUMA systems must provide partitioning of data objects that adapts quickly to changing workloads by employing efficient load balancing algorithms. Moreover, by means of data and thread placement, the storage engine must minimize remote memory accesses by primarily working in data object partitions that are located in the local main memory. In turn, this leaves sufficient link capacities for remote accesses caused by inevitable communication during query processing and by load balancing operations.

3.2 Classification of DBMS Architectures

In this section, we classify existing database system architectures including their respective query processing models and primarily compare them for their ability to scale up on large NUMA systems and to allow fine-grained adaptations at runtime. Additionally, we compare the DBMS architectures for the remaining requirements of an energy-aware database system from the previous chapter and introduce the core concepts of our *Living Partitions* architecture.

As we have shown in Section 3.1, scalability on large scale-up systems is mainly a matter of a local data object access (R-03) as well as a latch-free data object access (R-04), because remote memory accesses are costly on NUMA systems and latches (including atomic instructions) do not even scale on a single processor when being frequently accessed by different hardware threads in parallel. To enable fine-grained runtime adaptations, we have to consider more requirements. For instance, to enable *Resource Adaptivity*, the DBMS architecture mainly needs to be elastic in terms of work to hardware thread assignment (R-05), which is the antagonist of an always local data object access (R03) that is needed to provide scalability. Nevertheless, to enable *Storage Adaptivity*, a latch-free data object access (R-04) is once again beneficial, since latching is the natural blocker for physical storage transformations.

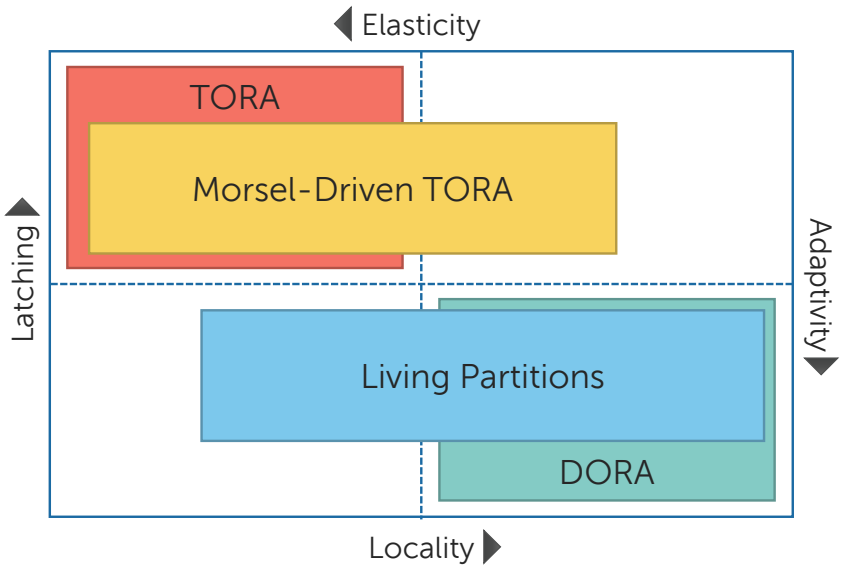


Figure 3.2: High-level comparison of DBMS architectures.

In Figure 3.2, we visualized the mentioned high-level characteristics, i.e., locality vs. elasticity and latching vs. adaptivity, and classify existing database system architectures as well as our *Living Partitions* architecture accordingly. Note that this classification only covers high-level aspects and we will give a more detailed classification in Table 3.3 considering all of our requirements. In the following, we will discuss the individual architectures and their ability to fulfill our requirements.

Table 3.3: Comparison of architectures regarding their ability to fulfill our requirements for energy awareness. Highlighted cells mark open points of the data-oriented architecture.

#	Requirement	TORA	Morsel-Driven TORA	DORA	Living Partitions
01	Main memory-centric	✓	✓	(✓)	✓
02	Scalability		○	(✓)	✓
03	Local data object access		○	✓	✓
04	Latch-free data object access			✓	✓
05	Flexible work to hardware thread assignment	✓	✓		✓
06	High degree of parallelism	○	✓	✓	✓
07	Small partitions		✓	✓	✓
08	Serialized partition access			✓	✓
09	Late binding of physical operators				✓
10	Flexible partitioning scheme		✓	✓	✓
11	Flexible partition to processor mapping		✓	✓	✓

We focus on the traditional *transaction-oriented* architecture as well as its *morsel-driven* extension and the *data-oriented* architecture including our *Living Partitions* extension.

3.2.1 Transaction-Oriented Architecture

The traditional transaction-oriented architecture (TORA) is the most widespread way of designing a database system. As depicted in Figure 3.3, TORA treats transactions as the first-class citizens, which corresponds to the natural anticipation of a human. Hence, transactions (the actual work) are directly executed by threads, which are assigned to physical hardware contexts. This assignment is either done by the operating system or by the DBMS itself. Since the basic transaction-oriented architecture does not enforce a local data object access on NUMA systems, this assignment is flexible and can easily be changed during execution. However, the references to non-local data object accesses mostly cause remote memory accesses on NUMA systems, which strongly limits the scalability of this architecture [106]. Moreover, transaction threads can access every available data object in the pool, which requires latches to protect simultaneously running threads against each other when reading or modifying a data object concurrently.

Thus, we consider TORA as an architecture that is highly elastic with a strong tendency to non-local data object accesses and high latching efforts according to our

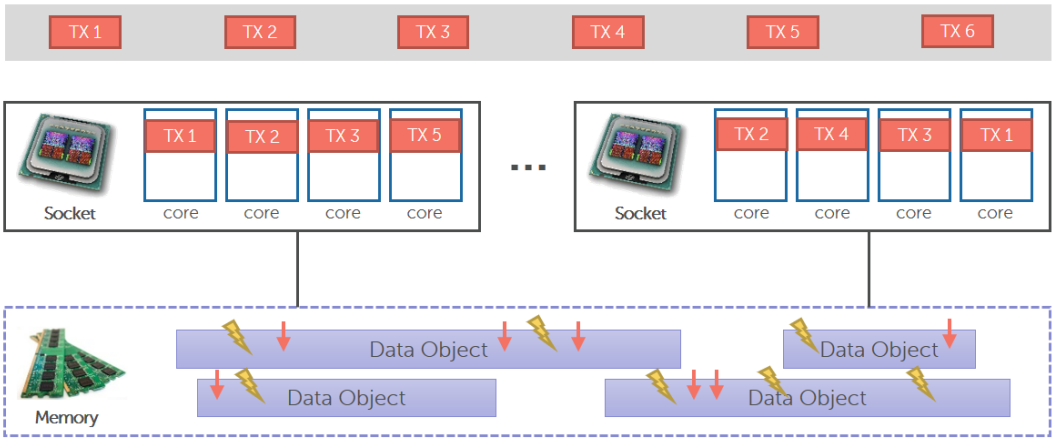


Figure 3.3: Query processing in a transaction-oriented architecture.

high-level classification in Figure 3.2. Regarding our requirements list in Table 3.3, we can conclude that the basic transaction-oriented architecture only fulfills the following requirements:

Main Memory-Centric (R-01). Transaction-oriented in-memory database systems were and still are subject of database research ranging from optimized data structures [79, 88, 115] over efficient snapshotting algorithms [97] to query execution plan (QEP) optimizations [113, 99]. The results show that the move from a disk-centric to a main memory-centric database system requires fundamental changes, but is not in conflict with the transaction-oriented architecture.

Flexible Work to Hardware Thread Assignment (R-05). Because the basic TORA approach does not enforce a local data object access (R-03), the transaction thread to hardware thread assignment can easily be adjusted at runtime.

High Degree of Parallelism (R-06). The basic transaction-oriented architecture is able to run multiple queries simultaneously (inter-query parallelism) as well as multiple independent operators in parallel (inter-operator parallelism). However, the basic TORA approach lacks the ability to internally parallelize single operators (intra-operator parallelism), which potentially leads to a small degree of parallelism when executing a small amount long running analytical queries.

To bypass most of the issues leading to a bad scalability behavior of the transaction-oriented architecture on modern massively parallel hardware and especially on NUMA systems, the morsel-driven TORA [87] approach respectively other partitioning-based approaches [110, 111] for the transaction-oriented architecture were proposed. The idea of morsel-driven TORA and similar solutions is to split data objects into

small morsels (partitions), which are placed in the local memory of a specific processor, and schedule plan operators as close as possible to the respective data location. Additionally, the approach tries to store intermediate results that occur during the query processing in the local memory of the processor the current operator is running on. However, a local data object access can not be guaranteed anymore when operating on those intermediate results or when communication between operators is required (e.g., in a join). Thus, the morsel-driven TORA approach is located close to the basic transaction-oriented architecture within our high-level classification (cf., Figure 3.2) with the difference that it has a stronger tendency to a local data object access, while still providing the same elasticity as the basic TORA. In terms of our requirements for an energy-aware database system, the morsel-driven TORA approach addresses the following requirements:

Scalability (R-02). As already mentioned, scalability on NUMA systems is mainly a matter of a local data object access (R-03) and a latch-free data object access (R-04). Thus, the TORA extension improves the scalability behavior, because the local data object accesses are more likely to happen. Nevertheless, the high latching efforts are still remaining.

Local Data Object Access (R-03). Since the approach tries to schedule plan operators as close as possible to the physical location of a morsel, the local data object access is significantly improved. However, communication between operators respectively operators that work on multiple morsels once again requires remote data object accesses. The same issue holds for operators that work on intermediate results that are scattered to multiple physical memory locations.

High Degree of Parallelism (R-06). The concept of processing morsels implicitly enables internal parallelism inside of a single operator and thus, the approach is able to reach a high degree of parallelism, since all levels of parallelism are now available staring from the coarse-grained inter-query parallelism to the fine-grained intra-operator parallelism.

Small Partitions (R-07). The concept of a morsel is equivalent to a horizontal partitioning approach. Nevertheless, the size of morsel is chosen small enough to be processable by a single hardware thread and thus, small partitions are available for fine-grained runtime adaptations. However, the morsel-driven TORA approach assumes an optimal partitioning of base data objects, which leads to an odd data distribution over the sockets of a NUMA system in case of a varying workload. Hence, this architecture also requires *Data Placement Adaptivity*, which is hard to achieve due to the high latching costs.

Flexible Partitioning Scheme and Partition Mapping (R-10/11) While the original morsel-driven TORA approach assumes a static partitioning, other works addressed this issue by proposing different online partitioning strategies.

The first approach [110] uses the `move_pages` functionality of Linux to transparently move physical pages of a data object to another processor. Another approach [111] employs an explicit partitioning to move and resize partitions at runtime based on the access pattern of operators. Moreover, both works also discuss the issue of scheduling operators close to the data depending on their memory access pattern. A drawback of those approaches is that partitions become inaccessible during the repartitioning process.

The basic as well as the morsel-driven transaction-oriented architecture support a multitude of query processing models that mainly differ in the type and the size of a chunk that is processed by a physical operator. The traditional tuple-at-a-time processing model follows the Volcano iterator model [45] where each operator pulls tuple by tuple from its preceding operators. This tuple-wise processing approach naturally fits to the row-oriented database systems of the early days. Due to the emergence of column-oriented DBMSs especially in the context of main memory database technology, the column-at-a-time processing model [22] gained a lot of attention. In this model, columns serve as input and output of single operators, because the base data objects in column-stores are organized column-wise instead of row-wise. Hence, the column-at-a-time processing model is the logical consequence for a column-oriented database system, which turned out to show significant performance increases for analytical workloads. An extension of the column-at-a-time approach is the vector-at-a-time processing model [24] that passes portions of a column between operators. The key concept is that those portions are small enough to fit into the processor cache to preserve locality when processing the same data by different operators. While the mentioned approaches pass raw tables/rows or columns/vectors between the operators of the query execution plan, the indexed table-at-a-time processing model produces only indexed tables as intermediate results to support succeeding operators in doing their respective work. The processing model is considered as a cooperative model, because the indexed columns of the intermediate results depend on demands of the consuming operator.

To summarize, the transaction-oriented architecture is the most widespread foundation of existing DBMS implementations and sees transactions as the first-class citizens supporting multiple query processing model flavors whose feasibility mainly depends on the physical organization of the storage (e.g., row-wise or columnar). As we have discussed, this architecture issues a lot of remote data object accesses and requires high latching efforts when accessing data objects in parallel, which leads to a bad scalability of large NUMA systems. Thus, the morsel-driven extension of TORA tries to improve those scalability limitations by increasing the number of local data object accesses, which is only possible to a certain degree. Based on our analysis, we conclude that even the morsel-driven TORA approach only fulfills a small set of our requirements for an energy-aware DBMS making it a bad candidate for further investigations.

3.2.2 Data-Oriented Architecture

The counterpart of the transaction-oriented architecture is the *data-oriented architecture (DORA)*, which turns the entire architectural paradigm upside down. For the data-oriented architecture, data is the first-class citizen instead of a transaction. As depicted in Figure 3.4, each data object is horizontally split into disjoint partitions (circles in the figure) and each of the partitions is placed in the local memory of a processor. To keep the access to those partitions local and latch-free, each partition is assigned to exactly one worker thread that can exclusively access the respective partition. A data-oriented database system typically uses as many worker threads as available hardware threads on the system (each pinned to a unique hardware thread) and thus, a worker thread is in charge of multiple partitions that belong to different data objects.

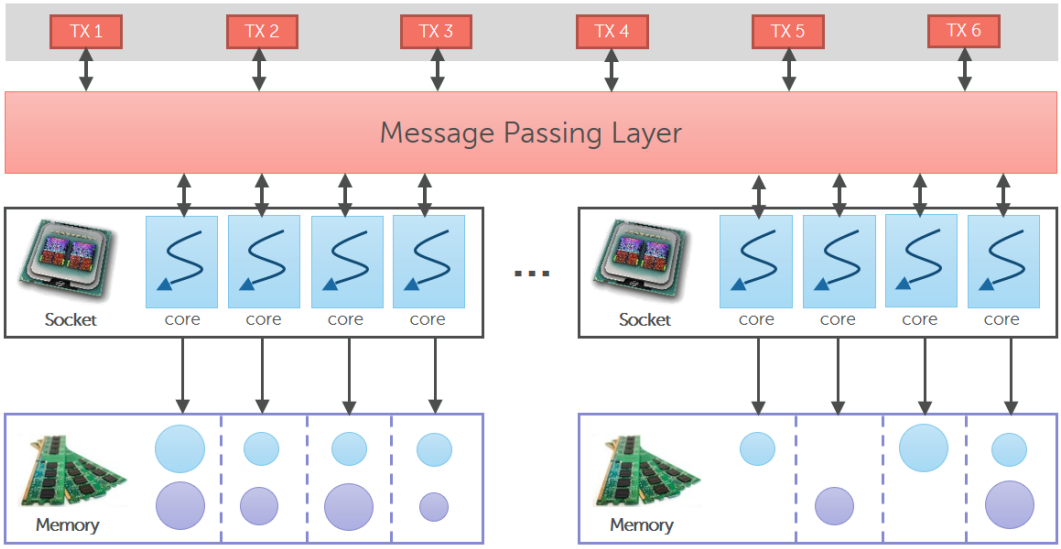


Figure 3.4: Query processing in a data-oriented architecture.

This kind of data-oriented transaction execution is similar to shared nothing architectures employed for scale-out scenarios where data objects (including intermediate results) are horizontally partitioned across multiple servers and a partition can only be accessed by server-local compute resources. The bottleneck in such a scale-out setup is usually the communication network between servers and thus, data object accesses should preferably be local. The same problem applies for NUMA systems except for the difference that the communication network between the individual processors is the bottleneck in such a scale-up architecture. Moreover, the data-oriented architecture for scale-up system goes even a step further and considers a single hardware thread instead of a processor or server as the smallest execution unit that is in charge of its exclusive partitions [104] (R-07). Using this fine-grained execution quantum, DORA addresses the problem of the bad scalability of latches

even inside of a single processor, because each partition can only be locally accessed by a single thread (R-03/04/08).

Nevertheless, the latch-free and local data object access comes for additional costs at other spots of the architecture. In opposite to the transaction-oriented architecture, the DORA approach needs an explicit communication facility to enable the message exchange between worker threads that effectively execute operators on their partitions. For instance, while a transaction in a TORA environment can simply latch and access a data object, a transaction in a DORA environment needs to issue messages to the responsible worker threads that finally access the data object. Hence, the data-oriented architecture requires a distributed query processing model. Another disadvantage of the data-oriented architecture is its sensitivity to workloads that do not fit to the current partitioning scheme, which results in an odd utilization of worker threads and a reduced query execution performance. To cope with this issue, DORA systems usually implement a *Data Placement Adaptivity* mechanism [104, 106] to adjust the partitioning scheme at runtime (R-10/11).

Based on the discussed characteristics of the data-oriented architecture, we place it in our high-level classification in Figure 3.2 in the opposite corner of the competing transaction-oriented approach, because data object accesses are always guaranteed to be local, which results in a bad elasticity, and data objects do not need to be latched, which makes DORA suitable for fine-grained runtime adaptations. Nevertheless, also the data-oriented architecture requires latches, which moved from the data object access layer to the message passing layer making it the most critical component within a DORA-based database system. Regarding our requirements list for an energy-aware DBMS, Table 3.3 shows that the data-oriented architecture already fulfills most of the requirements except for the following ones:

Main Memory-Centric (R-01). So far, the data-oriented architecture has only been investigated in the context of disk-based database systems [104]. While TORA as well as DORA-based DBMSs profit from the general advancements of general in-memory technology (e.g., cache optimized data structures), performance and scalability of a DORA-based system highly depend on the design of the message passing layer, which needs to keep pace with the increased data object access speed to avoid worker threads that starve from messages. This message passing layer has not been investigated so far and will be subject of our further considerations in the context of the data-oriented architecture.

Scalability (R-02). The data-oriented architecture has already been evaluated in the context of medium-scale (8 sockets) NUMA systems including the proposition of corresponding architectural concepts [106]. The evaluation revealed that the DORA approach exhibits an almost linear scalability on such scale-up systems. However, as we showed in our NUMA systems evaluation (cf., Table 3.2), communication related bottlenecks become even more critical when moving to large-scale NUMA systems. Additionally, the move from disk-based to an in-memory system increases to overall performance and thus, the pressure

on potential contention points. Hence, the scalability of the data-oriented architecture will be subject of our further investigations especially in the context of an in-memory database system.

Flexible Work to Hardware Thread Assignment (R-05). Compared to the transaction-oriented architecture, the DORA-approach imposes a static mapping of partitions to worker threads, which are pinned to a specific hardware thread. Since the execution of an operator, which does the actual work, is bound to a specific partition, the work to hardware thread assignment is statically fixed. Hence, the only option to deal with workload imbalances is to adapt the partitioning scheme of the data objects, which comes at high energy costs.

Late Binding of Physical Operators (R-09). Similar to the TORA approach, the data-oriented architecture for scale-up systems binds physical operators at query compilation time. However, because the message passing layer already decouples data objects accesses from the actual transaction execution, DORA provides a good foundation for fulfilling this requirement.

To summarize, compared to the transaction-oriented architecture, which was sufficient in times a low parallelism, the data-oriented architecture is designed for scalability and has already been investigated in terms of medium-scale NUMA systems and showed an almost linear scalability. Moreover, the DORA approach fulfills a lot of our requirements for an energy-aware DBMS making it an excellent foundation for our further considerations. However, DORA still misses some important requirements and lacks an investigation as well as appropriate concepts for scalability on large scale-up NUMA systems, which is mainly a question of the message passing layer especially in the context of in-memory database technology. We will address this specific topic in Section 3.3.

3.2.3 Living Partitions Architecture

In this section, we present our novel *Living Partitions* architecture, which is based on the data-oriented architecture and mainly enables fine-grained adaptivity at runtime. While the traditional transaction-oriented architecture assumes data objects as a passive lifeless state in main memory transactions are operating on, the data-oriented architecture already perceives data objects as something central and active that is composed of small partitions, which adapt its partitioning scheme according to the current workload.

The Living Partitions architecture goes an important step further and understands a partition of a data object as a living organism that needs to evolve over time to minimize its energy footprint by quickly responding to environmental influences by changing its location, size, or by specializing itself. Hence, the data-oriented

architecture is a good foundation for achieving this goal, because all data objects are implicitly partitioned into small chunks. However, DORA enforces a static partition to hardware thread mapping, which is a blocker for our *Resource Adaptivity*. For that reason, the Living Partitions architecture relaxes this mapping as shown in Figure 3.5. In the following we describe how this flexible work to hardware thread assignment (R-05) is achieved (the numbers of the items correspond to the numbers in the figure):

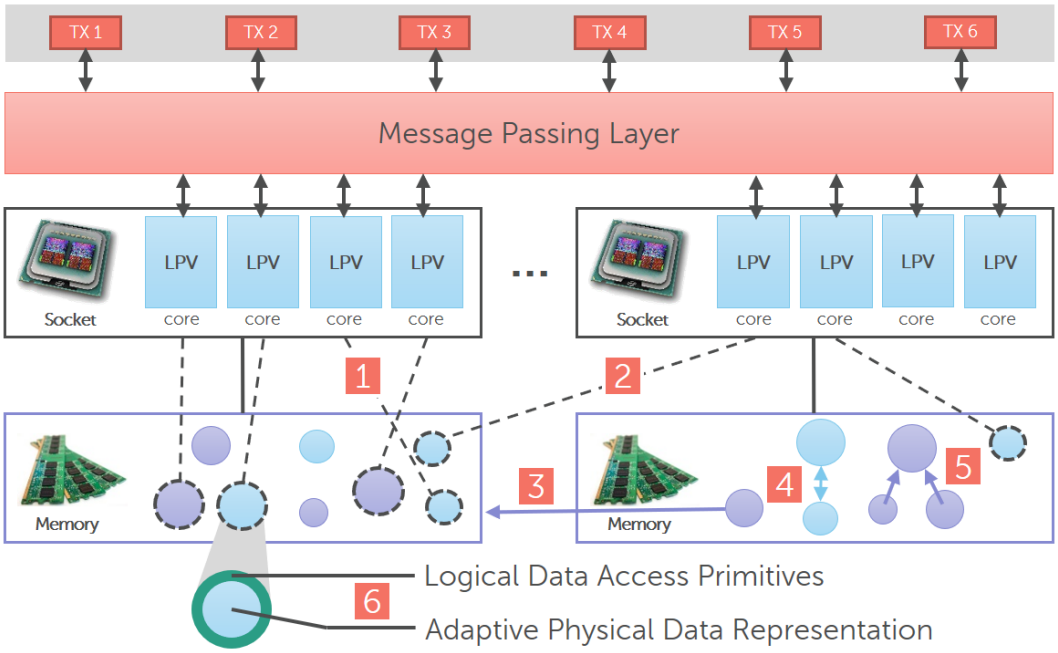


Figure 3.5: Query processing in the Living Partitions architecture.

(1) In the Living Partitions architecture, the Living Partitions are located in the local memory of a home processor. Instead of assigning partitions to a specific hardware thread of the processor, all hardware threads are able to process a partition in the local memory with the limitation that only *one thread processes a partition at a specific point in time* (R-08). Therefore, we run *Living Partition Vitalizers (LPV)* on each hardware thread of the respective processor that effectively bring partitions to life by granting compute resources to them. Thus, Living Partitions are latched very coarse-grained to enforce a serial access to them.

(2) To push elasticity respectively resource adaptivity even further, the Living Partitions architecture allows an LPV to process living partitions that are located in the local memory of a remote processor. Since resource adaptivity is a short-term measure, we use this remote processing mechanism to cope with temporary imbalances originating from a partitioning that does not fit

the current workload demands. Conceptually, this mechanism is equivalent to the morsel-driven TORA approach.

The next feature set addresses the requirements of *Data Placement Adaptivity*, which requires a flexible partitioning scheme (R-10) as well as a flexible partition to processor mapping (R-11). The original DORA approach already includes adaptive data placement facilities in terms of a variable partitioning scheme. Nevertheless, the Living Partition architecture goes a step further and is able to regulate the degree of parallelism at runtime, which is mainly a dependency of the number of living partitions a data object is split into. Moreover, entire living partitions can be physically moved between the local main memories of the individual processors. In the following, we describe the three mechanisms in detail:

(3) The first mechanism regarding data placement adaptivity is the ability of the architecture to physically migrate living partitions at runtime between the main memory of single processors (R-11). As we already outlined, migrating living partitions induces additional costs in terms of resource usage and energy consumption, because data needs to be physically copied and is thus considered as a long-term measure. Moreover, a living partition can not be modified during this process, which negatively affects query processing performance. However, since a living partition only comprises a small amount of the actual data object, this process can happen on a partition by partition basis resulting a low impact on the actual query processing throughput.

(4) The next mechanism of the Living Partitions architecture allows living partitions to be split into multiple child partitions at runtime. This process effectively results in an increase of the number of living partitions a data object is split into. Hence, the possible degree of parallelism can be increased using this process as long as the previous living partition count of a data object was lower than the total number of LPVs. Moreover, the mechanism of splitting living partitions is used to distribute hot spots, for instance, if a living partition is accessed unproportionally often.

(5) The last mechanism regarding data placement adaptivity is the merge operation, which is the reversal of the split operation. The merge operation effectively reduces the possible degree of parallelism, which is typically not desired on massively parallel hardware. Nevertheless, the merge has its right to exist, because a high number of living partitions induces higher costs in terms of infrastructure and maintenance (e.g., message passing costs and partition table size). Hence, this mechanisms makes sense for data objects that are not frequently accessed compared to others. Moreover, the combination of the split and merge mechanism actually enables a flexible partitioning scheme (R-10).

The remaining requirement for an energy-aware database system that is missing in the TORA as well as the DORA approach is the late binding of physical opera-

tors (R-09), which mainly addresses *Storage Adaptivity*. Since the Living Partitions architecture perceives a living partition as an autonomous organism that can change its physical data representation at any point in time, the storage layout-dependent physical operator can not be determined at query compilation time, because the physical representation may be changed between query compilation and the actual execution. Furthermore, the physical data format can be different across the individual living partitions. Hence, our last mechanism addresses exactly this issue:

(6) To cope with this issue, the Living Partitions architecture imposes an indirection layer between the logical operator and the actual physical data representation. Thus, a living partition exposes logical data access primitives such as a scan (with filter), an insert, and a delete. The actual decision on which physical operators are used for the data access are solely up to the living partition itself. For instance, a logical filtered scan operation can be physically executed using an index scan on living partition A and using a column or row scan on living partition B.

To classify our *Living Partitions* architecture in the high-level comparison in Figure 3.2, we place it close to the data-oriented architecture in the bottom right corner. Nevertheless, the Living Partitions architecture is able to provide a better elasticity compared to the basic DORA approach, which comes at the cost of additional latching costs, because partitions are not statically mapped anymore to a specific thread. However, this additional latching is very coarse-grained on a per-partition level and is thus, not very frequently invoked. Regarding our complete list of requirements for an energy-aware DBMS, the Living Partitions architecture fulfills all of the requirements assuming scalability on large scale-up NUMA systems in the context of in-memory database technology, which we will investigate in Section 3.3.

To summarize, our *Living Partitions* architecture is based on the data-oriented architecture and inherits its superior scalability characteristics, which still need to be proved on large-scale NUMA systems especially for in-memory database systems. Our extensions mainly comprise the necessary requirements for enabling fine-grained adaptivity at runtime such as the flexible work to hardware thread assignment and the late binding of physical operators on a per-living partition basis.

3.3 DORA for In-Memory DBMSs on Large-Scale NUMA Systems

The research question we want to answer within this section is whether the data-oriented architecture is able to scale up on large-scale NUMA systems (R-01) such as the SGI UV family [124] presented in Section 3.1 especially when being employed in a main memory-centric database system (R-02). Hence, we will design and evaluate

a prof of concept of our in-memory database system *ERIS* that focuses on the basic data object access primitives (i.e., the full table scan and the index scan) and implements a data-oriented architecture. As already outlined, the message passing layer is the most critical component of such an architecture, because the local data object accesses need to amortize the additional messaging costs. Moreover, we will prototypically investigate the load balancing component, to prove the overall feasibility of *Data Placement Adaptivity* in such an environment.

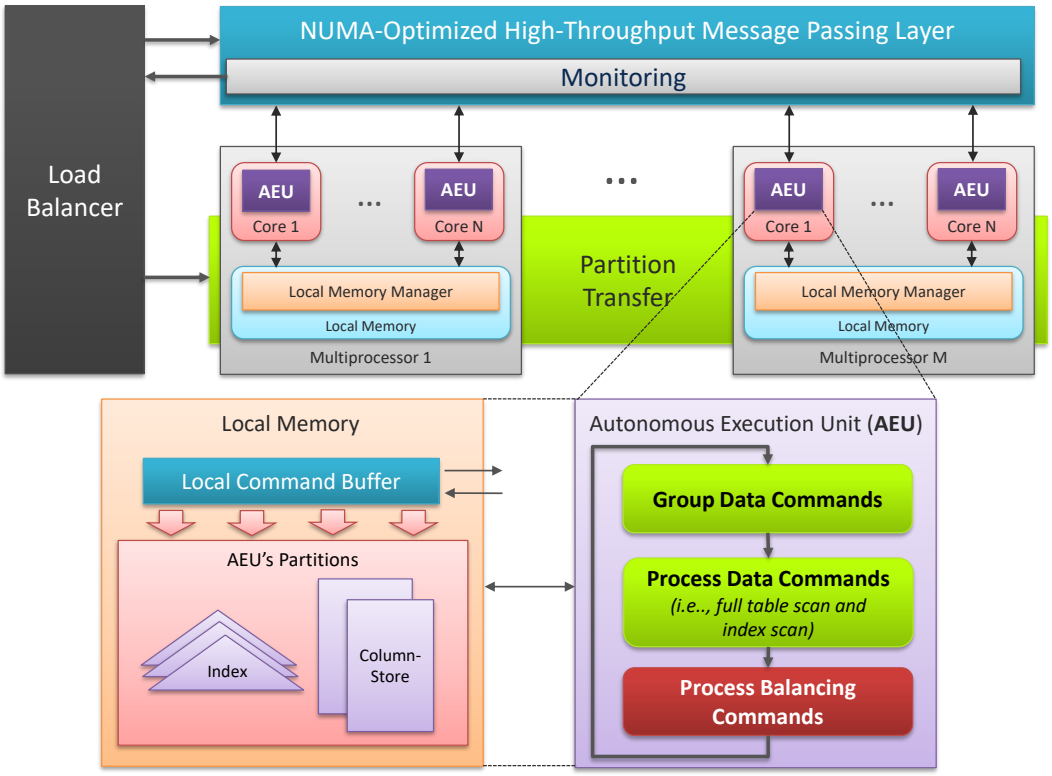


Figure 3.6: Architectural overview of the ERIS PoC and AEU details.

We start with a description of the architecture of our ERIS proof of concept (PoC) and its individual components as visualized in Figure 3.6. The central components of the PoC are the worker threads, which we call *Autonomous Execution Units (AEU)*. Each core, respectively hardware context, of the platform runs exactly one AEU. All AEUs pinned on the same multiprocessor use a common memory manager, because they share the same local main memory and are thus able to quickly exchange data partitions during load balancing. A set of partitions – each belonging to a different data object – is evenly assigned to each AEU. The AEU's main task is to manage its partitions and to process incoming data commands (i.e., full table scans and index scans) on these partitions. To efficiently route data commands during query processing between AEUs, our PoC includes a NUMA-optimized high-throughput message passing layer. The NUMA-aware load balancer of the PoC

observes the current load of the AEUs via a monitoring component and triggers balancing commands in case of an odd AEU utilization. In the following we describe the central components of our PoC in more detail.

3.3.1 AEUs and Memory Management

Traditional architectures (i.e., TORA) bind transactions to a number of threads and use a global memory manager (per data object). This way of accessing and storing data is highly discouraging when running on NUMA platforms, because data is distributed in an uncoordinated way across the memory of the different multiprocessors. In turn, this causes a high number of remote memory accesses by the transaction threads that are accessing the data object.

For that reason, our PoC employs a data-oriented architecture where each data object is logically partitioned. Each available core of the system runs an AEU, which is bound to be only executed on this single core or hardware context respectively. Every single AEU gets assigned a set of disjoint partitions and is exclusively responsible for that portion of the data object. This approach restricts memory accesses of an AEU to the multiprocessor's local main memory and data objects do not have to be protected against concurrent accesses via latches. Our PoC primarily uses range partitioning to split data objects into partitions. We decided against hash partitioning, because it is not order preserving and thus disallows efficient range scans and hinders an efficient load balancing. In scenarios where a table is solely completely scanned, we employ physical data size partitioning instead of range partitioning, because there is no suitable attribute as partitioning criteria available. Here, our ERIS PoC only keeps track of those AEUs that actually store a partition of the corresponding data object and uses the multicast capabilities of the message passing layer to distribute data commands.

Regarding memory management, a global memory manager (per data object) is not feasible on a NUMA platform. Instead, our PoC employs one memory manager per multiprocessor and data object (i.e., a table or index). Per-multiprocessor memory managers help to reduce the contention on the memory management subsystem, which is often the bottleneck during write operations to a data object. Moreover, this approach limits allocations of AEUs to the local main memory and enables the load balancer to perform an efficient intra-node balancing. To scale with a high number of cores per multiprocessor, our memory managers use thread-local caching mechanisms and thus, decrease contention on the local memory management.

At the bottom of Figure 3.6 we illustrate the AEU loop as well as the local memory organization of an AEU. The AEU mainly keeps local data command buffers and the actual data object partitions, which are either stored as a column-store for a full table scan or as an index for point queries). In the first stage of the loop, the AEU scans its data command buffer, which is periodically filled by the message passing layer, and groups commands by the accessed data object and the command type. This optimization step is beneficial to coalesce the same type of access to the same partition. Following the grouping step, the AEU actually processes its data

command buffer, which is the most time consuming part of the loop. Afterwards, the AEU checks its command buffer for pending balancing or transfer commands. Such commands force an AEU to grow or shrink its partition or to transfer a range of its partition to another AEU. We discuss the details of the load balancing process in Section 3.3.3.

3.3.2 NUMA-Optimized High-Throughput Message Passing Layer

The message passing layer is the most essential component of our ERIS PoC, because AEUs have to be supplied with data commands just in time. Especially during the execution of analytical queries, large amounts of data commands have to be routed between AEUs (e.g., lookup operations during a join). Thus, the main goal of the message passing layer is to distribute data commands at a high throughput. A data command consists of a storage operation type (i.e., full table scan or index scan), a data object identifier, a reference to a callback function, a data segment that contains all the necessary parameters for the storage operation (e.g., a batch of keys for the lookup or filters for an index scan), and additional data that is necessary for the query processing. Our message passing layer is shown in Figure 3.7. The core components are the partition tables, which keep track of the partitioning scheme of individual data objects. As already mentioned, a data object is either clustered, respectively sorted, on one or more of its attributes or it is distributed without any partitioning criteria. In the clustered case, the routing table stores the attribute range to AEU mapping (range partition table). If the data object is not partitioned on any attribute, the routing table only saves whether or not an AEU stores a partition of that data object (bitmap partition table). Since the routing tables are small data structure that are rarely updated (only during load balancing) and are frequently read, they usually fit into the caches of all multiprocessors and are thus not causing any remote memory accesses.

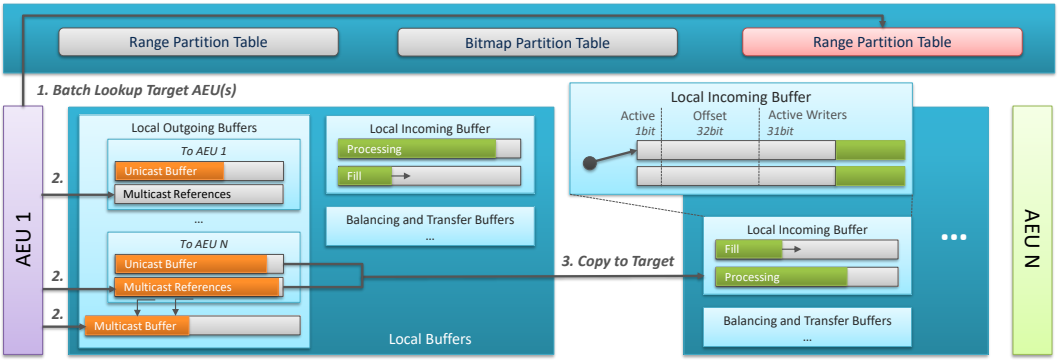


Figure 3.7: NUMA-optimized high-throughput message passing layer.

Besides the routing tables, our message passing layer uses a comprehensive buffering strategy to reduce the contention and to increase batch sizes. Each AEU uses a set of outgoing buffers – one unicast buffer and one multicast reference buffer for

each running AEU in the system – , a multicast buffer, and two bigger incoming buffers. All buffer types are stored in the local main memory of each AEU to provide fast access to them.

Every time an AEU generates a data command during the processing stage, it starts with a batch lookup of the responsible AEUs for that data command in the corresponding routing table of the target data object (step 1 in Figure 3.7). The routing tables use the content of the data segment of the data command to lookup the designated target AEUs. As soon as the target AEUs are determined, the routing layer splits the command into smaller pieces, for instance if a lookup data command contains keys in its data segment that belong to different partitions. Data commands for a single AEU are written to the corresponding outgoing buffer of the source AEU (step 2). If multiple AEUs are responsible for a data command (e.g., a full table scan that needs to be distributed to different AEUs), the command itself is written to the multicast buffer and references to this data command are stored in the individual multicast reference buffers. If an outgoing buffer is either full or the AEU starts over its processing loop, the specific outgoing buffer including its multicast data commands is copied to the incoming buffer of the target AEU (step 3). This local pre-buffering dramatically increases the data command routing throughput, because the contention on the incoming buffers is reduced and multiple data commands can be copied sequentially. Thus, the high latency of remote memory accesses on the NUMA platform does not become the bottleneck.

While outgoing buffers are private to an AEU and thus, do not require any concurrency control, incoming buffers are written by different AEUs and are read by the host AEU at the same time. Hence, incoming buffers need an efficient and lightweight concurrency control mechanism. We employ an adapted version of the lightweight multi-buffer proposed in LLAMA [89]. Each AEU has two incoming buffers of an equal size. One buffer is currently writable for all AEUs and the other one is currently the processed data command buffer of the owning AEU. To implement incoming buffers using a lightweight latching, each of them contains a 64bit wide buffer descriptor that uses 1bit for determining whether the buffer is still active or not, 32bit to save the current offset inside the buffer, and the remaining 31bit for storing the number of active writers to the buffer. If an AEU wants to write to an incoming buffer, it first determines the writable buffer, increases the offset by the size of data that needs to be written, increments the number of active writers, and finally atomically updates the buffer descriptor using an atomic compare-and-swap instruction. If the atomic buffer descriptor update fails, the entire process is repeated. This approach allows multiple AEUs to write to the incoming buffer in parallel. After the AEU has successfully written its data commands, it atomically decrements the number of active writer to the buffer. The owning AEU of the incoming buffers swaps both buffers each time it enters the data commands processing stage. The AEU updates the pointer to the new writable buffer and atomically flips the active bits of both buffers. Afterwards, it holds on until all AEUs have finished writing their data to the new data command processing buffer.

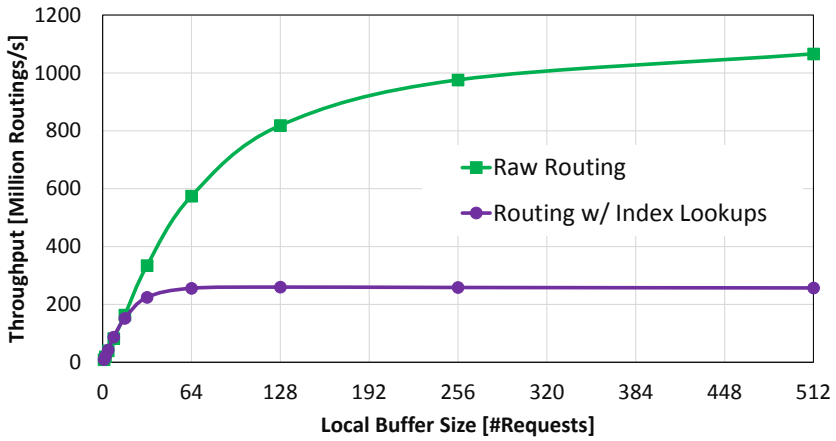


Figure 3.8: Message passing throughput as a function of the local buffer size on an 8-socket NUMA system.

We evaluated the effect of the outgoing buffer size on the routing throughput on the AMD machine (see Section 3.1) and visualize the results in Figure 3.8. Regarding the raw routing throughput, where AEUs skip the processing phase, we observe that the throughput doubles with the size of the outgoing buffers until the bandwidth of the NUMA interconnects start to saturate. If we enable the processing phase and generate index lookup data commands, the peak throughput is already reached for an outgoing buffer size of 128 data commands, because the throughput is now dominated by the index lookups during the processing stage of the AEUs. A comparison of both measurements demonstrates the effectiveness of our NUMA-optimized high-throughput message passing layer.

3.3.3 Load Balancing

Besides the message passing component, our PoC includes a NUMA-aware load balancer component to evaluate the *Data Placement Adaptivity* mechanism in case of changing workload that does not fit the current partitioning scheme. The main objective of the load balancer is the maximization of parallelism. We distinguish between two major scenarios:

(1) The data object is always scanned in its entireness and is thus not partitioned by a specific attribute.

(2) The data object faces lookups or scans in certain ranges and is thus partitioned by one or more attributes.

In the first case, the physical partition size is the considered metric for the load balancer, because the scan works only efficient, if all AEUs have to scan the same amount of data. In the second scenario, we use the access frequency as primary metric, because lookups and range scans only involve a certain set of AEUs. Additional

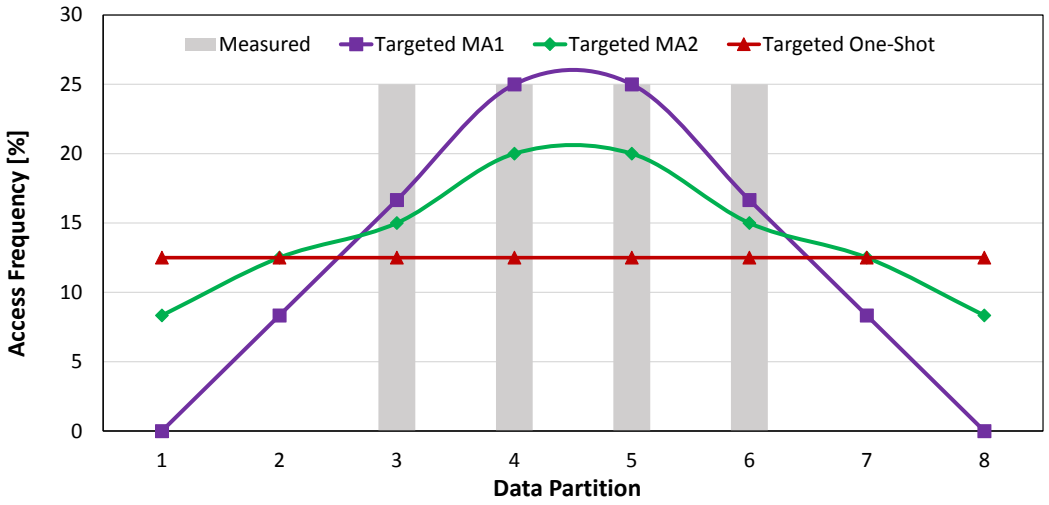


Figure 3.9: Configurable load balancing algorithm of our ERIS PoC.

metrics for the latter scenario are the mean execution time of a data command for a specific partition. Different execution times are mostly a result of different depths of tree-based index structures, or column store partitions that are frequently accessed but fit into the cache of a multiprocessor, or effects of data command coalescing.

The adaptation loop of our PoC starts with the monitoring of the different metrics on a per partition level. Based on the captured metrics, the load balancer periodically checks the load for imbalances. If the standard deviation between the different AEUs exceeds a given threshold, the load balancer executes a load balancing algorithm that calculates a new target partitioning. With the help of the current and the targeted partitioning, the load balancer computes a series of balancing commands that are routed to the involved AEUs. Such balancing commands include the new data respectively key range the AEU is responsible for and a set of transfer commands that instruct the AEU where it has to fetch the missing partition data from. Next, we will describe our configurable load balancing algorithm and the NUMA-aware partition transfer mechanisms in more detail.

Configurable Load Balancing Algorithm

The load balancing algorithm receives the approximated metric distribution of a single data object of the recent sampling period as well as the current partitioning as input and outputs the targeted partitioning for the respective data object. Figure 3.9 shows an exemplary metric distribution measurement (the access frequency in this specific case) that was sampled per partition and is represented as a histogram. In this specific scenario, partitions 3 to 6 each received 25 % of the accesses, which is a severe imbalance. The most aggressive, but also most expensive approach is taken by the *One-Shot* load balancing algorithm configuration. This algorithm configuration computes the average access frequencies of all partitions and calcu-

lates a target partitioning that is fully balanced. The *One-Shot* configuration is suitable for workloads that change rarely but heavily. An alternative configuration uses the moving average (*MA*). For instance, the *MA1* configuration computes for each partition the moving average of the partition's direct neighbors including itself and adjusts the target partitioning appropriately using a series of merge and split operations. The *MA* configuration adapts more slowly to the new workload, but does not cause as much balancing overhead as the *One-Shot* algorithm and is thus suitable for highly dynamic workloads. As depicted in Figure 3.9, the aggressiveness of the *MA* configuration depends on the range the moving average is calculated over and turns into the *One-Shot* algorithm when configured as *MA7* in our setup, because it equally calculates the full average across all partitions. As soon as the load balancing algorithm has finished the calculation of the target partitioning, the latter is compared to the current partitioning and the load balancer generates a series of balancing commands for that data object.

NUMA-Aware Partition Transfer

If the load needs to be balanced, each AEU that has to split or merge its local partition of the data object receives a balancing command. Such a balancing command first includes the new partition ranges for the AEU. The AEU updates the corresponding routing tables and saves the lower and upper bounds of its ranges internally, because it has to compare each incoming data command against its bounds to check its validity. If the AEU encounters an invalid data command (i.e., a data command that references keys outside its updated range) it forwards this data command to the AEU that is now responsible for the range. If a data object is balanced that is not partitioned by a specific criteria, the balancing command includes the number of tuples that have to be fetched or handed over to another AEU. To avoid situations of overlapping partition ranges, all AEUs that are involved in the current balancing cycle have to be shortly synchronized for updating a data object's routing table.

Besides the information about the new partition ranges, a balancing command includes a set of transfer commands. We continue with the example of Figure 3.9 and look at the balancing of partitions 1 to 4 using the *One-Shot* load balancing algorithm. The balancing of partitions 5 to 8 is very similar, because this specific workload is symmetric. In Figure 3.10, we illustrate the corresponding partition transfer process for that example. For reasons of simplicity, we assume a NUMA system consisting of four multiprocessors and two cores per multiprocessor in the example. Because the current range of partitions 1 and 2 is not accessed by the new workload, partition 1 receives a first transfer command instructing the AEU to take over the entire range of partition 2. Since both partitions reside in the same local memory and thus, in the same memory management domain, AEU 1 uses the cheap *link* mechanism to transfer partition 2. To do so, AEU 1 firstly unlinks the respective portion of the partition (the complete partition in our case). Afterwards, AEU 1 simply links (e.g., in case of a tree-based index) respectively appends (in case of a

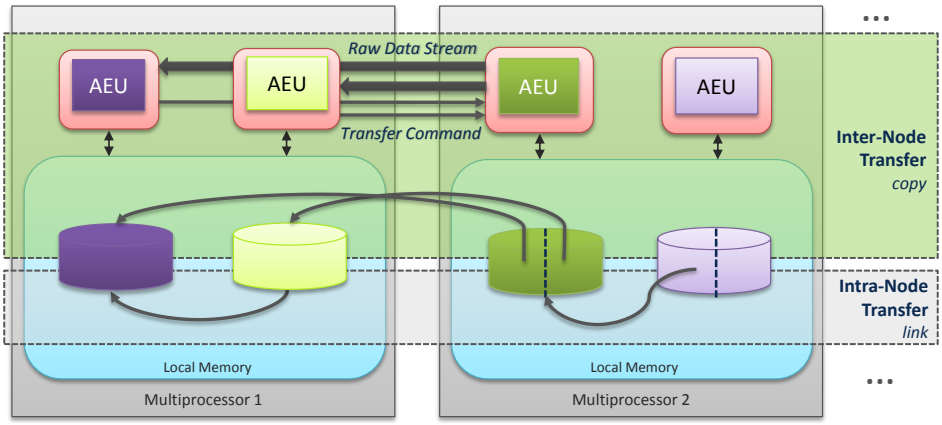


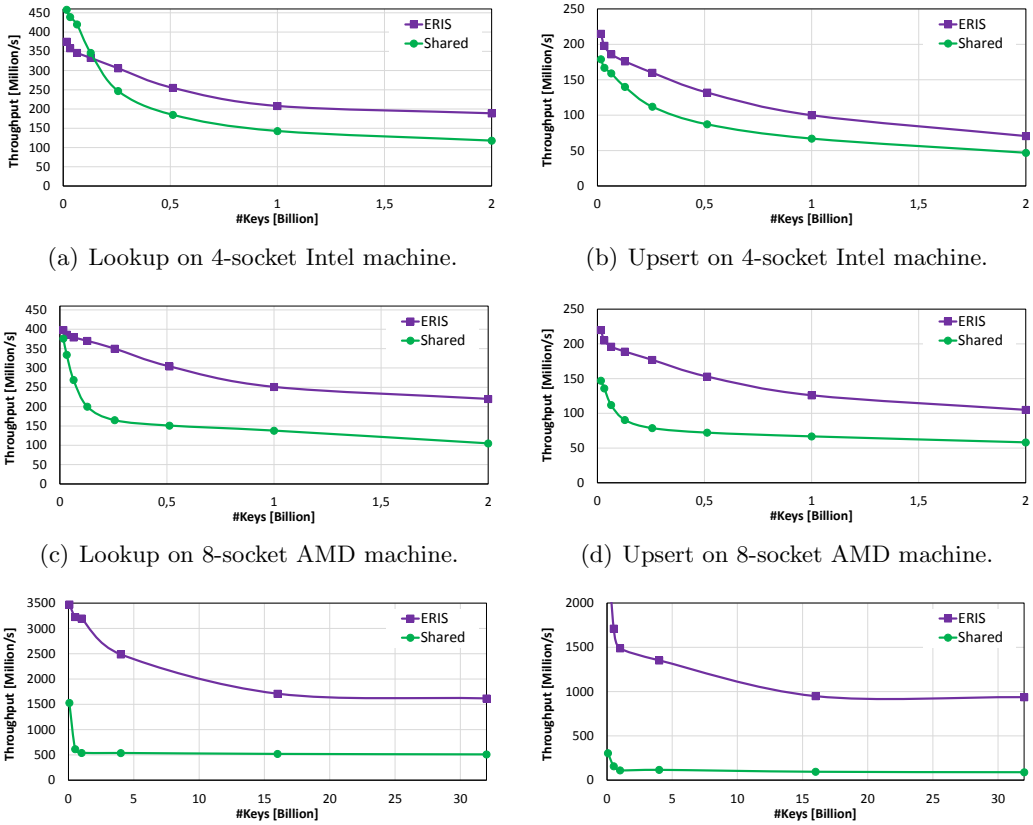
Figure 3.10: NUMA-aware partition transfer via link and copy.

column-store) partition 2 to its own partition 1. For the transfer of half of partition 4 to partition 3, our PoC also uses the *link* mechanism, because both partitions are located on multiprocessor 2. The remaining two transfers from partition 3 to partitions 1 and 2 are inter-node transfers and thus use the *copy* mechanism for the partition transfer. Such a *copy* operation requires a cooperation between source and target AEU, if the data object is stored as an index to avoid the high latency of remote memory accesses while traversing through the tree-based index. In this case, the source AEU forwards the transfer command to the source AEU, which flattens the partition to an exchange format and streams it sequentially to the target AEU. The target AEU converts the data stream back to an index and links it to its existing partition. If the data object is already stored in a flat format such as a column store, the target AEU directly copies the data from the source AEU. As soon as an AEU has processed all its transfer commands, it becomes ready to continue normal operation and when all AEUs have completed their balancing command, the balancing loop starts over again.

3.3.4 Implementation Details and Evaluation

In this section, we detail our implementation of an AEU and investigate the behavior of our ERIS PoC. We evaluate the PoCs' scan, lookup, and upsert performance by comparing it to the NUMA-agnostic shared index respectively shared scan as baseline, which corresponds to the basic transaction-oriented architecture. For the baseline experiments we use the same data structures as for the AEUs. The difference is that those data structures are not partitioned and are thus, synchronized via atomic instructions for updates, because they are accessed by different transaction threads in parallel.

An AEU implements a simple column store as well as a prefix tree [20] as index. We decided to use a prefix tree, because this index structure is order-preserving (applies not to a hash table), in-memory optimized, and offers a high update performance



(a) Lookup on 4-socket Intel machine.

(b) Upsert on 4-socket Intel machine.

(c) Lookup on 8-socket AMD machine.

(d) Upsert on 8-socket AMD machine.

(e) Lookup on SGI UV 2000.

(f) Upsert on SGI UV 2000.

Figure 3.11: Lookup/Upsert throughput depending on index size.

(does not apply to a B+-Tree). To implement the range partition tables of our PoC, we decided to deploy a CSB+-Tree [116], because it works fast for sparsely distributed data and it scales with an increasing number of ranges, respectively AEUs, compared to a simple array.

For our evaluation, we compare different configurations (e.g., different index sizes) and reason about the observed results. Furthermore, we compare our ERIS PoC to different memory allocation strategies for column data and evaluate their scan performances. For certain experiments, we additionally present results of hardware event measurements to gain deeper insights in the algorithms' behaviors. Moreover, in Figure 3.8 we presented experiment results that show that the NUMA-aware high-throughput routing is not the bottleneck of our PoC.

Evaluation Setup

All our experiments are executed on the three machines that were introduced in Section 3.1.2, i.e., the 4-socket Intel machine, the 8-socket AMD machine, and the

SGI UV 2000 (cf., Table 3.1). The executable files are compiled with the `g++` compiler, using optimization level `O3`. The shared index experiments are executed with `numactl -interleave=all` to interleave the memory across all available multiprocessors. Interleaving the memory resulted in slightly higher throughputs of the shared index compared to memory agnostic executions. A single benchmark run for upsert/lookup performance comprises two phases; (1) random keys are inserted into the index for about one minute and afterwards, (2) random keys are read from the index for another minute. Insert and lookup throughputs are reported for the two phases respectively. In the static workload cases, keys are uniformly distributed across the dense key domain. If not stated otherwise, the prefix trees are configured with a prefix length of 8bit. In a scan performance benchmark run, a column with random entries is generated and afterwards scanned repeatedly for one minute.

The throughput of storage operations is measured and reported by the application itself. We use different tools to measure the utilization of the links that connect multiprocessors, the open source tool `likwid` [2] on the Intel and the AMD machine and the SGI tool `linkstat-uv` on the SGI machine. Hardware performance counters are evaluated with `likwid` on the Intel and the AMD machine and `VampirTrace` [5] on the SGI machine.

Static Workload Experiments

For all static workload experiments, the load balancer and thus, the data placement adaptivity, is deactivated and the workload does not change over the whole benchmark run, i.e., keys to upsert or lookup are evenly distributed across the key domain and scans are over the full key domain.

Point Access with Different Index Sizes. In the first set of experiments, we compare the lookup and upsert throughput of the PoC to the shared index for different index sizes. The results are shown in Figure 3.11. The index sizes vary from 16 million keys to 2 billion keys on the AMD and Intel machine and from 16 million keys to 32 billion keys on the larger SGI machine. As a reference, 1 billion keys require approximately 25 GBs in memory and in our largest experiment, the index is as big as 0.8TB. Figure 3.11(a) shows that for small indexes on the small machine, the shared index outperforms our DORA PoC. The reason for this is the small overhead that is introduced by the NUMA-optimized message passing layer. However, as the number of multiprocessors increases and with larger indexes, our PoC clearly supersedes the shared index. While on the eight node AMD machine, the PoC has a throughput that is about 1.6 times higher than the shared index (Figure 3.11(c), 1 billion keys), on the larger SGI machine, our ERIS PoC executes already 3.5 times as many lookups per second as the shared index (Figure 3.11(e), 16 billion keys). Finally, Figure 3.11 shows that the upsert performance behaves similar to the lookup performance, except the lower absolute throughput values.

Scan Performance. In the second experiment, we compare our PoCs' scan performance with two different other memory allocation strategies for column data. The

results on the SGI machine are shown in Figure 3.12. In the experiment, all AEUs or parallel threads respectively[2] scan a column with about 8 billion entries. The memory for the column data is allocated (1) on one single multiprocessor (*Single RAM*), (2) interleaved on all multiprocessors (*Interleaved*), or (3) on the multiprocessor where the AEU is executed (ERIS PoC). In the *Single RAM* case, the scan performance is bound by the read bandwidth of the memory controller (cf. Table 3.2). The scan performance with interleaved memory is bound by the different link bandwidths. Only our data-oriented PoC is able to achieve optimal scan performance. Figure 3.12 shows that the PoC achieves a 6.6 times higher bandwidth than does reading from memory that is interleaved over all multiprocessors. In the past, interleaving has often been proposed as a method of choice to overcome NUMA effects. However, our experiment clearly shows the drawback of such an approach.

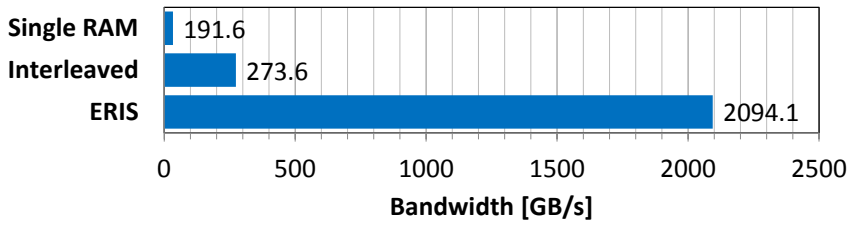


Figure 3.12: Scan bandwidth of our ERIS PoC compared to naïve memory allocation strategies on SGI UV 2000.

L3 Cache Usage. To better understand the lookup performance of our PoC and the shared index, we investigate the L3 cache usage in this experiment. For smaller index sizes, larger portions of the upper levels of the prefix trees fit in the caches. For larger index sizes, the last level cache plays a minor role and the performance is memory bound. It can be seen in Figure 3.11 that for increasing index sizes, the performance of the shared index is earlier memory-bound than the performance of the PoC. The explanation is that our DORA PoC makes better use of the L3 cache. Because each AEU in our ERIS PoC serves a distinct partition and hence a subset of the tree, there is better data locality and less concurrency for the L3 cache. Consequently, the upper levels of the tree fit in cache for larger index sizes (see, e.g., [76] for details on cache concurrency effects).

To verify our theory, we have calculated the L3 cache miss ratio for different index sizes on the AMD machine[3]. Furthermore, we have evaluated the state of the cache line for each L3 cache hit (for availability of the respective counters, this is evaluated on the Intel machine[4]). Figure 3.13 shows that the shared index causes

[2] 488 cores, or 61 multiprocessors, is largest possible working set in the batch system on our SGI machine.

[3] The L3 cache miss ratio is calculated as the quotient of the following hardware counters: `L3 Cache Misses` and `Request to L3 Cache` [9].

[4] The cache line states are measured using the `LLC_HITS` counter extensions in the C-Box of the Intel CPU [67].

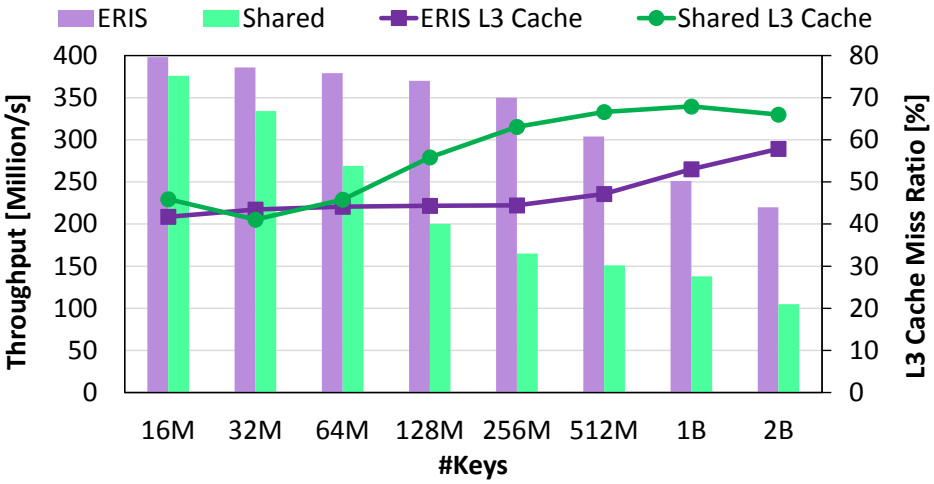


Figure 3.13: L3 cache miss ratio on AMD.

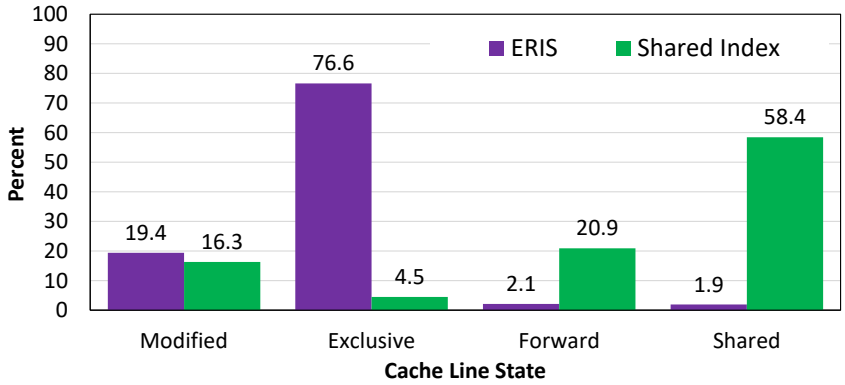


Figure 3.14: L3 cache line states on Intel - percentage of all hits (1B keys).

a higher miss ratio for smaller indexes, compared to our PoC. This is supported by Figure 3.14, which shows that for the shared index 79.3% of all hits are on *Shared* or *Forward* cache lines which implies that the same cache line is present in another cache. Cache lines that are kept in multiple caches reduce the effective size of all caches and increase the miss ratio. Our PoC on the other hand has significantly better data locality, which can be seen in Figure 3.14 where 97 % cache hits go to cache lines in *Modified* or *Exclusive* states.

Link and Memory Controller Usage. Our ERIS PoC is designed such that it reduces communications between multiprocessors. The significantly higher throughputs of our ERIS PoC as well as the L3 cache usage already suggest that this goal is achieved. However, to further verify our theses, we measure the average link us-

age over all links in a 10 seconds steady state window[5]. The results for the AMD machine are shown in Figure 3.15. The shared index has to transfer a total of 83.8 GB/s to fulfill all remote memory requests. At the same time, our PoC only transfers 17.8 GB/s (mainly caused by the data command routing facility) while at the same time achieving a higher throughput and thus performing more memory operations. The shared scan with interleaved memory allocation transfers a total of 75.6 GB/s, compared to 1.2 GB/s transfered by the PoC. These numbers together with the low level results in Section 3.1 explain the low throughput of the shared setup. Each remote access suffers from the worse latency and bandwidth compared to local memory accesses.

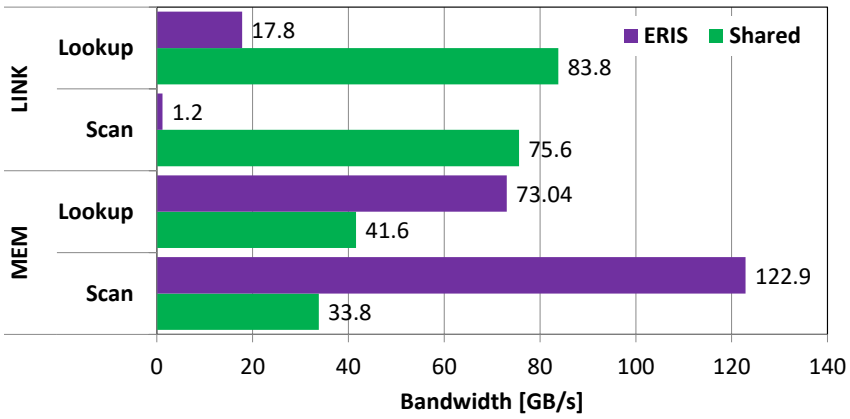


Figure 3.15: Link and memory controller activity on AMD (scan: 8 GB, lookup: 1 B Keys).

Together with the link utilization, we have measured the transfer bandwidth of the memory controllers[6]. The results are also shown in Figure 3.15. On average, only 1 out of 8 memory requests of the shared setup that go to local memory and remote memory requests suffer from higher latencies. Therefore, the shared index can only issue enough memory requests to transfer an average of 41.6 GB/s from all memory controllers while our PoC is able to transfer 73.0 GB/s. The shared scan produces a transfer rate of only 33.8 GB/s, compared to 122.9 GB/s transferred by the PoC. The transfer rate of our PoCs' scan operator equals 93.6 % of the possible accumulated memory bandwidth of the system (cf. Section 3.1).

Scalability. The main objective of our data-oriented PoC is to evaluate the scalability of the DORA approach on large-scale NUMA systems employing in-memory database technology. Hence, we conducted scalability measurements on the largest NUMA system available to us (the SGI machine) and show the respective results in Figure 3.16. The chart includes scalability measurements for the TORA approach

[5]We measure the link usage by evaluating the `Link Transmit Bandwidth` counters of the AMD CPU [9].

[6]We measure the memory controller by reading the `DRAM Accesses` counter of the AMD CPU [9].

(shared access) and the DORA approach (ERIS PoC) each for the full table scan and index lookups. The table scan executes column scans with a data size that is large enough to be fully memory-bound and the lookup operations access and index filled with 1 billion keys. The core allocation strategies differ for the scan and the lookup. While the scan allocates the cores evenly across the sockets, the lookup operation first fills a socket before a core on the next socket is allocated.

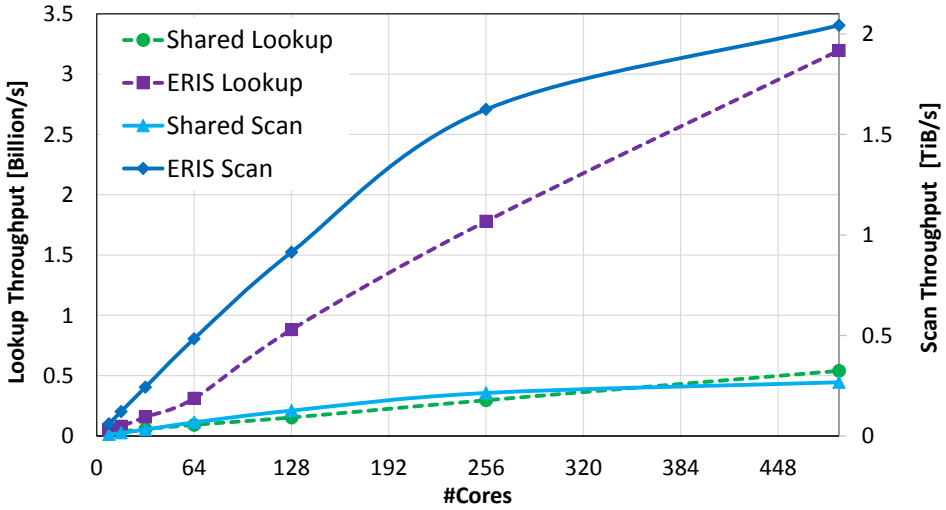


Figure 3.16: Scalability comparison of our ERIS PoC to the TORA approach for scan and lookup operations.

We observe that the NUMA-aware DORA approach shows a significantly better scalability behavior for both operations compared to the NUMA-agnostic TORA approach resulting in an about 6x higher overall throughput. The scan performance scales linearly when adding more cores until the memory controllers of the individual processors start to saturate and the scan is finally reaching almost the peak memory bandwidth of the machine. Due to the core allocation strategy used by the lookup operation, we measured a superlinear scale-up. This behavior is a result of the better cache utilization in the data-oriented architecture, which allows a higher portion of the index data structure to fit into the cache when more processors are activated.

Dynamic Workload Experiments

In this section, we show that our PoC is able to keep a high throughput even under changing workload conditions. For our experiments, we use a workload that randomly accesses the full key range (lookup) of 512 million keys for an initial period of 10 seconds. After this period, the workload changes drastically such that only half of all keys (in the range from 128 M to 384 M) are accessed afterwards. In the remaining time of the experiment, the workload is changed 4 more times with 20

seconds between any two changes. These remaining changes are only slight changes which are simulated by shifting the key range of interest by 8 million to the left.

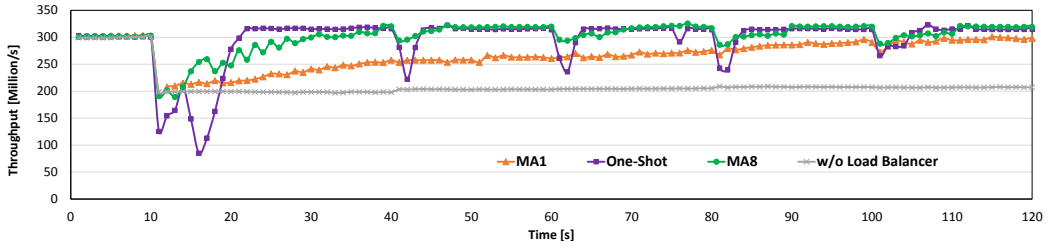


Figure 3.17: Load balancer experiments on 8-socket AMD machine.

Figure 3.17 shows the lookup throughput of the ERIS PoC over time. The above described workload changes are easily recognizable as short drops in the throughput curve. The chart contains performance numbers for a baseline run without load balancer and for three different load balancing algorithms (cf., Section 3.3.3), the One-Shot algorithm as well as Moving Average algorithms with window sizes of 1 and 8. The One-Shot algorithm causes the deepest drop of the throughput after each workload change, because all repartitionings that are necessary to regain a fully balanced workload are executed at once. This causes large overhead (some partitions need to be copied) but at the same time results in the fastest recovery time. The chart shows that the throughput reaches its maximum again shortly after each workload change. The other extreme is the MA1 algorithm, which only slightly adapts the partitioning in each evaluation period. Hence, the performance does not drop that drastically, but it takes more time before the maximum throughput is reached again. The MA8 algorithm appears to be the best compromise in this setup between performance drop and recovery time on that specific system.

As a conclusion, we note that the MA load balancing algorithm, with a parameter that depends on the machine, offers the best performance. Moreover, the parameter can be used to shift the behavior between gentle performance drops and quick recovery times, depending on the constraints of the application running on top.

3.3.5 Summary and Conclusions

The overall intention of this section was to investigate the scalability behavior of the data-oriented architecture on large-scale NUMA systems in the context of a main memory-centric database system to check the ability of DORA to fulfill our requirements R-01 (main-memory centric) and R-02 (scalability) as shown by Table 3.3. Moreover, our goal was to validate the feasibility of *Data Placement Adaptivity* in such an environment. Hence, we designed a corresponding PoC that implements the data-oriented architecture including the full table scan and the index lookup as the two primitive in-memory operations as well as appropriate online load balancing mechanisms. Since lookup operations are orders of magnitude faster compared a full table scan, we focused on the message passing layer, which is needed by the

data-oriented architecture to allow a local access to the actual data object. Thus, our main research question was whether the local data object accesses amortize the additional message passing costs to bypass the increased remote main memory access latency especially induced by latency-bound operations such as the index lookup. As we have shown in our evaluation, this question can be answered positively when employing our NUMA-aware high-throughput message passing layer. Furthermore, we proved that the data-oriented architecture exhibits a superior scalability even on large-scale NUMA machines (R-02) when using in-memory data structures (R-01). As conclusion we can emphasize that the DORA approach fulfills the two requirements under investigation and additionally allows efficient *Data Placement Adaptivity*.

3.4 ERIS Data Management System

In this section, we present our in-memory data management system *ERIS*, which is tailor-made to scale up on large-scale NUMA systems and to enable fine-grained adaptations at runtime and thus, allows us to implement and evaluate our *Energy Awareness by Adaptivity* concept for an energy-aware DBMS. In the previous section, we empirically proved scalability and the feasibility of *Data Placement Adaptivity* of the data-oriented architecture using our in-memory DORA proof of concept (PoC). However, this PoC was limited to a full table scan and an index scan as data object access primitives and was not capable of processing comprehensive queries and to ensure transactional properties. Moreover, the PoC implemented only the basic DORA approach.

Hence, we will focus our discussion in this section on the architectural changes that are necessary to implement our *Living Partitions* architecture, which also includes a major conceptual redesign of the message passing layer to deal with the additional freedom that was granted to the individual partitions by this extension of the data-oriented architecture (R-05). Furthermore, we will discuss the topic of the query processing model and transaction processing as well as related components of ERIS such as memory management, *Tasks*, and *Dataflows*. Finally, we will evaluate our ERIS implementation mainly in terms of scalability using a set of micro benchmarks and the TATP benchmark [62] on a large-scale SGI UV 3000 system and compare the results to our previous PoC.

3.4.1 Architecture

ERIS occupies all available physical compute resource (hardware threads) by running Living Partition Vitalizer (LPV) software threads on them. Each LPV is pinned to a distinct hardware thread. One of the main contributions of the Living Partitions architecture is to allow a flexible work to hardware thread assignment (R-05). Hence, LPVs are able to slip into multiple logical roles they are granting their physical compute resources to. Figure 3.18 depicts the corresponding architecture for a

single processor of the NUMA system. In the following, we will describe the three logical roles each LPV can slip into:

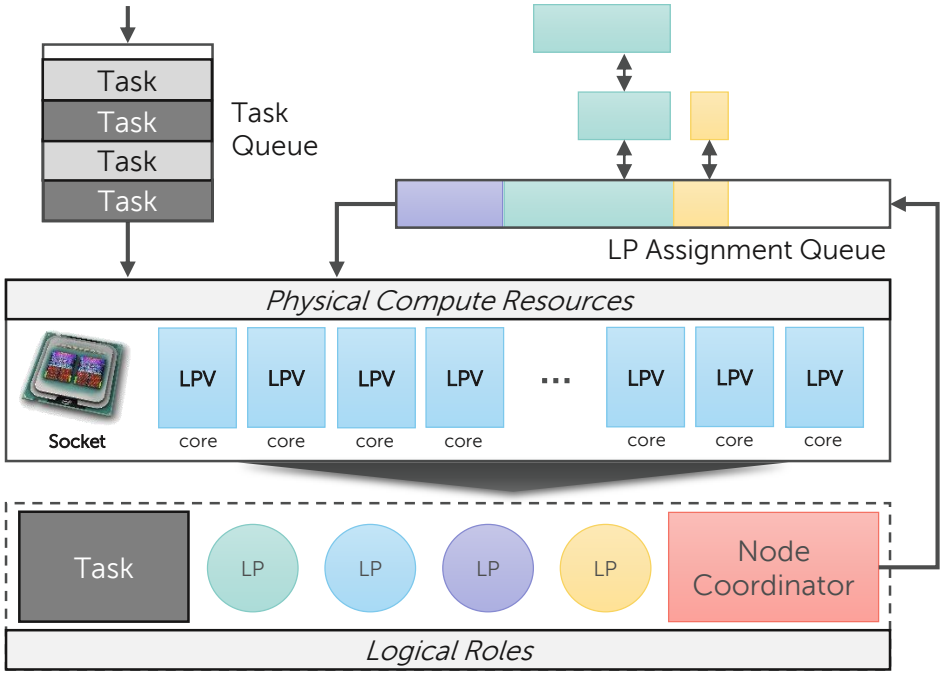


Figure 3.18: ERIS processing architecture of a single socket.

Task. ERIS tasks represent the sequential portion of a query such as query compilation and the controlling of the query execution plan, which consists of dataflows. We will present the details for tasks and dataflows in Section 3.4.2 respectively 3.4.3. A task in ERIS is externally scheduled and is bound to a specific socket of the NUMA system using the processor-specific task queue (cf., Figure 3.18). Hence, LPVs are able dequeue tasks from this queue and grant time of their respective compute resource to them. Such a task can be interrupted at any point in time by blocking operations, e.g., waiting until the execution of a dataflow finishes. In this case, the task processing is suspended and it is enqueued in the task queue as soon as the blocking operation finishes. ERIS tries to execute a task always on the same socket to limit its memory accesses to the local main memory.

Node Coordinator. Besides the task role, the node coordinator is another new logical role introduced by ERIS. While tasks are available at a varying amount and each of them can be processed in parallel by a single LPV, the node coordination role exists only once per socket. Thus, the node coordinator role can only be executed by one LPV at the same time. The job of the node coordinator is to manage central processor-local duties as well as the

management of the inter-processor communication. Most of the time, the node coordinator manages central jobs of the message passing layer, for instance, filling and maintaining the Living Partitions Assignment Queue. We will give additional details on this topic in Section 3.4.4.

Living Partition (LP). To process messages sent to a specific *Living Partition*, an LPV needs to slip into the role of that specific LP. Similar to the task queue, each socket employs an LP Assignment Queue that contains message buffers for the individual living partitions on that socket. Buffers for the same LP are grouped by the node coordinator to minimize the contention on this queue, because a living partition can only be processed by a single LPV at the same time. Hence, an LPV dequeues a set of buffers from the LP assignment queue, locks the corresponding living partition and processes the message buffers in the context of the respective LP. If new message buffers arrive during the processing phase of a living partition the node coordinator appends the new message buffers to the buffer set that is currently processed by an LPV.

Algorithm 1 LPV loop

```
1:  while not halt do
2:      ...                                                  // Task Queue
3:      while task ← dequeueTask() do
4:          processTask(task)
5:      end while
6:      ...                                                  // LP Assignments
7:      while lpAssignments ← dequeueLP() do
8:          lock(lpAssignments.lp)
9:          while buffer ← nextBuffer(lpAssignment) do
10:             processLP(lpAssignment.lp, buffer)
11:         end while
12:         unlock(lpAssignments.lp)
13:     end while
14:     ...                                                  // Node Coordinator
15:     if try to lock Node Coordinator then
16:         process Node Coordinator
17:         unlock Node Coordinator
18:     end if
19:     ...
20: end while
```

Each LPV executes the loop shown in Algorithm 1 to poll the task queue, the LP assignment queue, as well as the node coordinator latch to take over a specific role. We decided against an event-based system to avoid costly system calls that are additionally limited in their scalability. The outer loop (line 1–23) repeats the entire

procedure as long as the LPV is running. In the first step of this procedure, the LPV tries to dequeue a task from the task queue (line 4). As long as this operation succeeds, because tasks that are ready to be processed are present, the LPV slips into the role of the respective task and processes it until the task is finished or suspended (line 5). In the next step the LPV tries to dequeue an LP buffer set from the LP assignment queue (line 9). Every time this operation returns a buffer set, the LPV latches the referenced living partition (line 10) and processes the respective buffers in the role of that LP (line 11–13). Afterwards, the latch on the living partition is released (line 14). Finally, the LPV tries to acquire the node coordinator latch (line 18). If the lock acquisition succeeds, the respective LPV takes over the role of the node coordinator (line 19) and releases the node coordinator latch (line 20). The process repeats over and over again. To assist overutilized sockets in case of a bad partitioning, LPVs of a remote processor can be ordered by the energy-control loop (cf., Section 2.5.2) to additionally check that task queue and the LP assignment queue of that sockets. This access to remote data structures only happens explicitly on-demand to keep the contention of the processor-local queues and latches as low as possible.

Memory Management

Another critical and interesting topic on massively parallel NUMA systems is the memory management. Thus, ERIS needs a memory management subsystem that scales, considers locality, has a low contention, and obeys the characteristics of the DBMS architecture. Because NUMA systems have a local main memory per processor, it is a natural decision to employ a hierarchical approach for the memory management as visualized in Figure 3.19. The basic unit of the ERIS memory management are *Memory Managers* that use a thread-local allocation mechanism where a small set of the free memory is buffered per thread to reduce the contention on the management data structures.

At the highest level of the memory management is the *Global Memory Manager* that uses an interleaved memory allocation strategy, which distributes the physical memory allocation equally across all sockets of the machine. The global memory manager hosts the memory of data structures that need to be available on all processors of the system (e.g., meta information of data objects or partition tables).

At the next level of the hierarchy is the *Node Memory Manager*, which is responsible for managing solely the local main memory of a specific socket. This memory manager is configured to only allocate or recycle entire chunks of memory pages, because it only acts as a mediator between system calls (e.g., mmap) and depending memory managers. Depending memory managers are the persistent, the task, and the living partitions memory managers that differ in the lifetime of the allocated memory. The *Persistent Memory Manager* allocates its memory from the node memory manager and is used for persistent allocations such as internal data structures of the processor-local LPVs as well as shared data structures. In contrast, the *Task Memory Managers* are instantiated per task and are completely wiped as soon

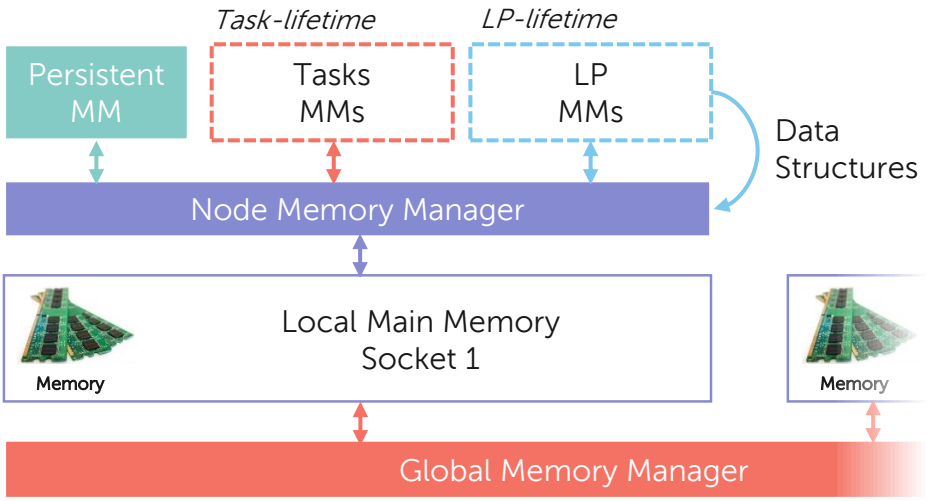


Figure 3.19: ERIS memory management.

as a task finishes to avoid memory leaks and costly deallocation calls, which typically amount to a high number. Another class of depending memory managers are the *Living Partition Memory Managers* that are created per living partition and host all internal data structures of an LP, which are wiped at the end of the LP lifetime. Moreover, living partitions are able to instantiate additional memory manager for storing the actual data of an LP, e.g., an index or a column store. Those memory managers also use the node memory manager to allocate entire chunks of memory pages and are at the latest destroyed when the corresponding living partition is destroyed.

3.4.2 Tasks

An *ERIS Task* is a piece of code that is sequentially executed by a Living Partition Vitalizer and is able to build and execute dataflows (cf., Section 3.4.3), to start and control transactions (cf., Section 3.4.5), as well as to create and destroy data objects. To implement a task, ERIS provides the *ERIS/C++ Framework* that includes all the necessary classes (e.g., Transaction and DataFlow). A task itself is not able to directly access data objects and thus, its execution amounts only to small amount of the query. Hence, tasks are not executed by heavyweight kernel mode threads in ERIS. In contrast, a task is executed using lightweight user mode threads (i.e., boost coroutines [82]) that save and restore CPU registers as well as the floating point unit (FPU) state in user mode when a task is executed, suspended or resumed, since tasks are able to execute blocking operations.

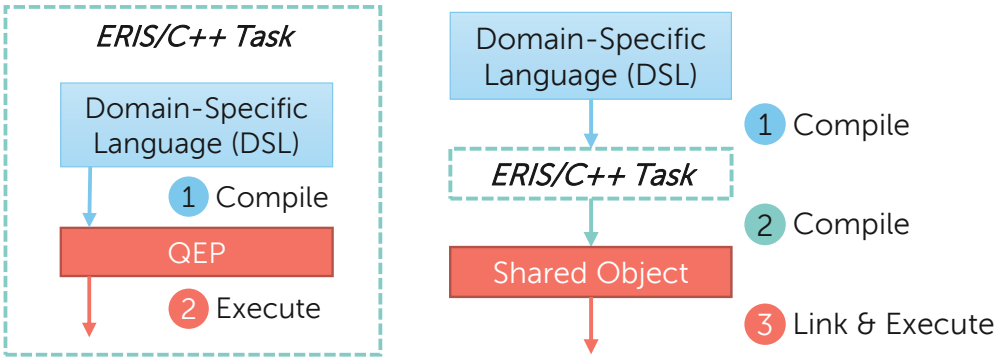
In Algorithm 2, we give a small example of an ERIS/C++ task to demonstrate the API. First, the task starts a new transaction by instantiating a `Transaction` object (line 1) followed by the creation of a `RelationalContainer` that stores the meta information for a new table (line 2). Since this table is allocated on the transient

Algorithm 2 Task example in ERIS/C++

```
1: Transaction tx;                                          // Start transaction
2: RelationalContainer container(true);                     // Create temporary relation
3: container.initialize()->await();                         // initialize container and await
4: MicroOperator op(container, [&](MicroOperator *op) -> void {
5:                                                          // Bootstrap code for root operator
6: }, [&] (ScanState *scanState) -> void {
7: ...                                                      // Actual data processing code
8: } );
9: DataFlow df(tx);                                         // Create dataflow
10: df.setRoot(&op);                                        // Assign root operator
11: df.await();                                             // Execute dataflow and await
12: tx.commit()->await();                                   // Commit transaction and await
```

stack, the table is destroyed as soon as the task finishes and is thus considered a temporary table. To create a table that survives the execution of the task, it needs to be allocated by the global memory manager (cf., Figure 3.19). In the next step (line 3), the new table as well as the partition table are physically created by calling the `initialize` method of the `container` object. Because this call physically creates the new table, messages have to be sent to the node coordinator, which actually creates the living partitions according to the partition table. This operation is asynchronous and hence, the task is blocked until the actual living partitions are created on all the sockets of the NUMA system. To free the LPV until the operation completed, the user mode thread is suspended in the meantime. Afterwards, the task starts to build a dataflow (line 9) that consists of multiple micro operators (line 4 and 10). The micro operator uses callback functions for sending the initial messages to the living partitions (only the root operator) and a second callback for the actual data processing. Once the construction of the dataflow is done, the dataflow is executed (line 11), which is also a blocking operation that needs to be awaited. Finally, the transaction is committed and awaited (line 12), too. All blocking operations return a `DataFlow` object that implements the `start` (start execution) and `await` (wait for completion) method, which can be called at any suitable point in time. Moreover, `DataFlow` objects can be grouped into a `DataFlowGroup` to wait for multiple dataflows to finish.

For static or precompiled queries, the corresponding ERIS/C++ task can be implemented by a programmer. Nevertheless, this way of implementing queries has multiple drawbacks: (1) It requires a deep knowledge of the ERIS/C++ API. (2) Implementing sophisticated queries is an extensive job. (3) No ad-hoc queries are supported. Hence, we propose two approaches for compiling and executing arbitrary queries that are depicted in Figure 3.20. In the following we will explain both approaches in more detail:



(a) Generic approach. (b) Compiler-based approach.

Figure 3.20: Query compilation approaches using *ERIS/C++*.

Generic Approach. The generic approach (Figure 3.20(a)) works similar to the traditional way of query compilation in database systems. First, the user specifies the query using a domain-specific language (DSL), like SQL. Afterwards, the query is compiled (step 1) to a query execution plan (QEP) where multiple generic operators are parametrized and are arranged in a tree that is finally executed by the database runtime (step 2). This process can be encapsulated in an ERIS/C++ task that does the query compilation including the optimization and outputs the QEP as a dataflow, which can be executed.

Compiler-Based Approach. The compiler-based approach (Figure 3.20(b)) requires a meta compiler that is able to compile the query into ERIS/C++ task code (step 1). In a second step this C++ task needs to be compiled into executable machine code (step 2). Finally, the resulting library (e.g., a shared object in Linux) is linked into the address space of the ERIS instance, which invokes the starting function for query execution (step 3). This way of query compilation is similar to just-in-time query compilation approaches as they are employed by state-of-the-art database systems [99, 56].

Both approaches face their respective advantages and drawbacks. For instance, the generic way on the one hand benefits from a fast query compilation, but may results in a bad query execution performance on the other hand, because operators are written in a highly generic fashion to deal with all possible execution cases, which leads to non-optimal machine code. The compiler-based approach faces exactly the opposite advantages and disadvantages. The time taken by the query compilation process increases, because of the additional compilation steps. However, the generated operator code is highly specialized, which amortizes the compilation costs for comprehensive queries.

3.4.3 Dataflows and Query Processing Model

To actually process and modify data objects, *ERIS* employs *Dataflows* that are executed in the context of a transaction. A dataflow is constructed as well as invoked by a task and consists of *Micro Operators*. Within a dataflow, such mirco operators are either statically connected as a directed graph or are loosely connected, if the actual control path of the dataflow depends on the processed data.

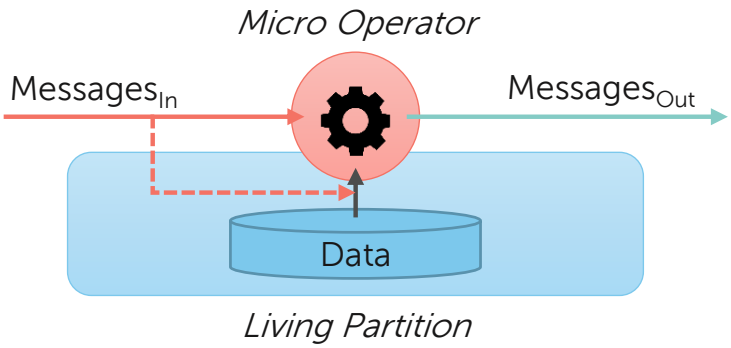


Figure 3.21: Micro Operator in *ERIS*.

In Figure 3.21, we visualized the basic processing model of a single micro operator. The micro operator itself is a piece of code that is executed within the context of a living partition, which contains the data. This piece of code is implemented as a callback function (cf., Algortihm 2 line 7) and is invoked by incoming messages. An incoming message defines a logical access primitive as well as auxiliary information to specify which portion of the living partition's data is of interest for the micro operator or how the data should be modified. For instance, the `UnorderedScan` is a logical access primitive for a relational table that lets the micro operator face every record of the table without any particular order. Using the auxiliary information, this scan can be restricted horizontally via a filter and vertically by defining the columns of interest. Because the logical access primitive does not enforce any ordering, micro operators can execute in parallel on all living partitions of the respective data object that potentially contain the data of interest, which is defined by the partition table (cf., Section 3.4.4). Such a logical access primitive does not specify any physical access paths, because physical operators are dynamically bound at execution time of a micro operator to fulfill requirement R-09 (late binding of physical operators), which is needed for *Storage Adaptivity*. We will discuss this topic in Chapter 5 in detail and give an overview of all logical access primitives required for relational query processing in ERIS. Besides consuming incoming messages, a micro operator can produce outgoing messages that invoke another micro operator for further query processing following the predefined connections of the dataflow or using a data-driven path.

To make the concept of dataflows and micro operators as well as the overall query processing model of ERIS more tangible, we will now demonstrate the query

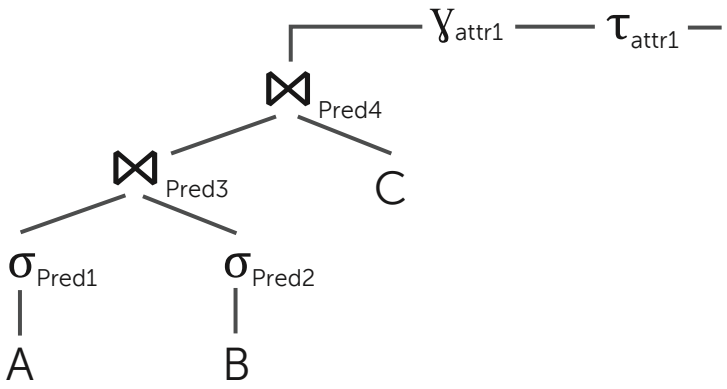


Figure 3.22: Logical query plan.

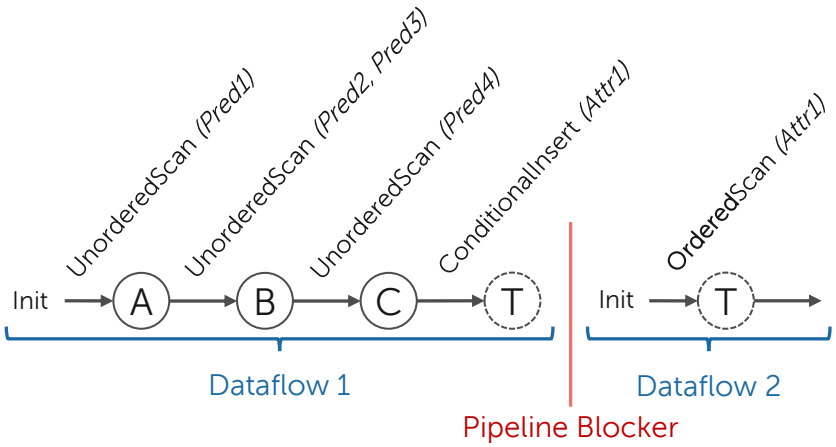


Figure 3.23: Macro query execution plan in *ERIS*.

compilation process for an exemplary query. The corresponding logical query plan of this query is depicted in Figure 3.22. This plan starts with a selection on table `A` and `B` using predicate `Pred1` respectively `Pred2` followed by a join of both results with predicate `Pred3` on table `C`. Afterwards, the result of this logical join is again joined with the entire table `C` using the join predicate `Pred4`. Finally the result of both joins is grouped (including the an aggregation function) and sorted by attribute `Attr1`.

A possible *Macro Query Execution Plan (MQEP)* of this query is visualized in Figure 3.23. This MQEP is not complete, because the physical access paths are not bound at this stage and the living partitions may need to reassemble the requested data of a micro operator using multiple physical operators respectively a local execution plan (cf., Chapter 5). Nevertheless, since the MQEP already consists of physical micro operators executable by the ERIS runtime, this execution plan is not considered as a logical one anymore.

Because ERIS employs an asynchronous and pipelined query processing model, pipeline blockers need to be identified first. In our specific example, the logical group and aggregate operator is breaking the pipeline, because the result of this operator needs to be materialized in the temporary table `T` before the succeeding sort operator is allowed to start. Thus, the ERIS task consists of two dataflows that execute one after another. At the beginning of `Dataflow 1`, an initial bootstrap message is sent to table `A` that issues an `UnorderedScan` with predicate `Pred1`. The corresponding micro operator is executed on all living partition that potentially include this data. Every time the micro operator is supplied with a qualified record, it sends a message to the succeeding micro operator operating on table `B`. This succeeding micro operator once again issues an `UnorderedScan` with the static predicate `Pred2` and the dynamic predicate `Pred3`, because this predicate depends on the tuple the previous micro operator faced before it sent the message. The same process repeats for the micro operator that operates on table `C`, which uses the dynamic predicate `Pred4`. At this stage, the micro operator on table `C` faces all tuple combinations that qualified for both joins. For all of those tuple combinations, the micro operator sends the respective message to another micro operator that operates on the temporary table `T` and issues the logical `ConditionalInsert` access primitive including the respective value of `Attribute 1`. This access primitive inserts the key (`Attr1`) value (tuple) pair into the temporary table `T`, if the key is not present or applies the aggregation function to the existing and the new value otherwise. To place the same keys in the same living partition, this operation requires the table to be partitioned by the key attributes. As soon as `Dataflow 1` finished, `Dataflow 2` is started by the task and executes the logical `OrderedScan` access primitive on the temporary table `T` by sending the initial messages. Contrary to the `UnorderedScan`, the `OrderedScan` enforces that the micro operator faces the requested tuples in a particular order. In our case, tuples are ordered by `Attr1`. Similar to the `ConditionalInsert`, the `OrderedScan` requires an appropriate partitioning.

The pattern of nested scans that is employed by `Dataflow 1` is similar to a physical nested-loop join operator. However, ERIS uses macro query execution plans and thus, the specific join algorithm can not be determined at query compilation time. For instance, if the living partitions of table `B` or `C` use an index instead of a full table scan for fetching the requested tuples for the micro operator an index or hash join algorithm is actually used. Since, such decisions are done on a per-living partition and can change during query execution, the actual physical join implementation can be mixed.

An important issue of ERIS' asynchronous query processing model is to determine when a dataflow actually finished its execution, because micro operators only see a limited scope of the whole dataflow. To cope with this issue, we use the following stop criterion:

$$\text{Messages}_{\text{Init}} + \sum \text{Messages}_{\text{Out}} - \text{Messages}_{\text{in}} = 0 \tag{3.1}$$

According to the criterion, each micro operator counts the number of incoming messages and the number of outgoing messages. As soon as the sum of all incoming and outgoing messages plus the number of initial messages reaches exactly zero, the dataflow finished, because no active messages are present for that dataflow anymore. From the implementation perspective, this mechanism is implemented using a counter variable that it atomically increased or decreased. Unfortunately, such a shared counter becomes a bottleneck on large-scale NUMA systems, because it is accessed frequently by multiple threads on different processors. Hence, LPVs in ERIS buffers the access to this counter as long as possible to reduce the contention.

3.4.4 Living Partitions-Enabled Message Passing Layer

In this section, we introduce the message passing layer of ERIS, which – compared to our PoC – is ready to support (1) comprehensive queries, (2) transactional properties, and (3) the living partitions architecture. To support all of those properties, the message passing layer requires a *sophisticated message format* that is able to associate a message to a dataflow and a transaction. Moreover, a message needs to carry the relevant query processing state (e.g., attribute values of previous micro operators), additional parameters for the respective logical access primitive, and needs to be space efficient. Besides changes of the message format, *architectural changes* to the message passing layer are necessary to enable the flexible living partitions to hardware thread mapping (R-05).

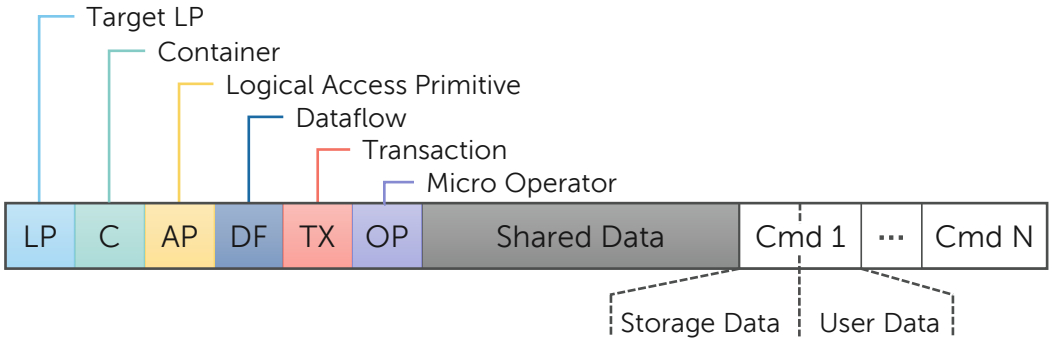


Figure 3.24: ERIS message format.

In Figure 3.24, we depict the message format employed by ERIS. As shown, the header of each message contains an 8 byte pointer to the target living partition or is set to `null`, if the message is a broadcast. The specific value of this field is determined by the partition table of the respective data object depending on the demands of the logical access primitive and its auxiliary parameters. Following the target living partition pointer, the message header includes a pointer to the data object (called container in ERIS), the logical access primitive, the associated dataflow, and the transaction context. Finally, the header includes a pointer to the callback function, which contains the actual micro operator code. Since ERIS is designed

to run on scale-up systems it is sufficient to store pointers in a message, because the cache coherence protocol fetches the actual object data transparently from the hosting processor, which saves copy costs ind reduces the message size. Similar to the partition table, such data structures are mostly read-only and fit into the cache to avoid message related bottlenecks. One exception is the message counter stored in the dataflow object, that needs to be updated frequently. However, as already mentioned we use sophisticated buffering strategies to avoid a high contention on this counter. Besides the header, a message includes a shared data field that includes micro operator-specific information shared across all of its messages (e.g., parameters). To reduce the memory footprint of messages and thus, the communication costs, the message passing layer tries to combine multiple messages sharing the same header into a single message. All of those sub messages (Command) follow the shared data field. Such a command is split into a storage data portion and a user data part. The storage data part contains parameters for the logical access primitive, for instance, the actual filters for an `UnorderedScan`. The information stored in the user data part is defined by the micro operator code, which issues the messages. For instance, the user data field may contains partial records from a previous micro operator.

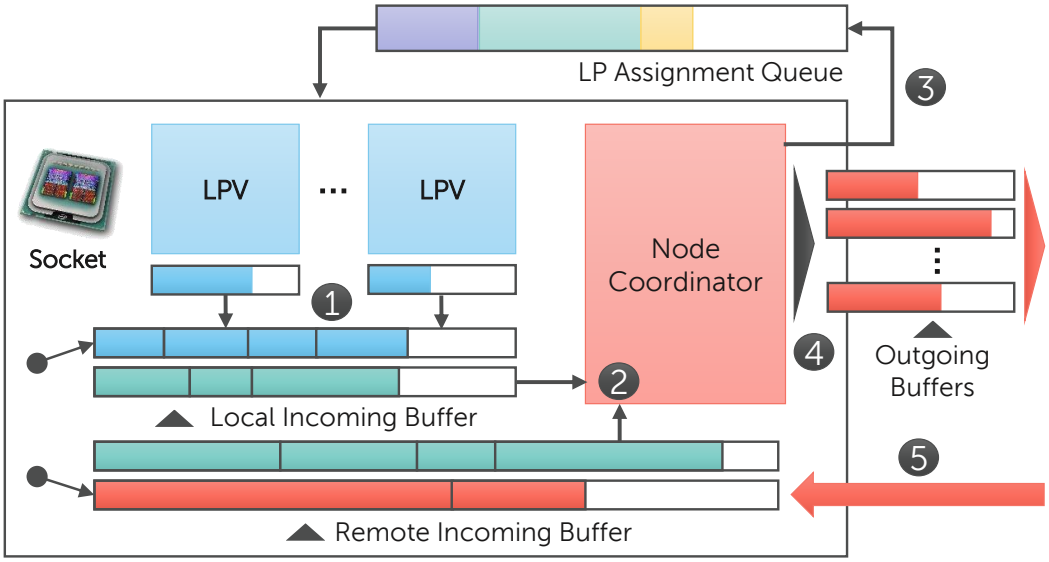


Figure 3.25: Living partition-enabled message passing layer in ERIS (socket-level).

In the following, we describe the architectural changes demanded by the living partitions architecture. The message passing layer of our data-oriented PoC directly exchanged messages between Autonomous Execution Units (AEU). In this architecture, each AEU requires as many outgoing buffers as AEUs running in the DBMS, which already leads to a quadratic complexity when increasing the number of AEUs and thus, hardware threads used. This complexity problem becomes even more severe when moving to the living partitions architecture, because partitions are not statically mapped to a single execution thread anymore. Hence, each liv-

Algorithm 3 Local/Remote buffer processing method of the node coordinator.

```
1:  function processBuffer(buffer)
2:      while message ← buffer.next() do
3:          if buffer.isLocal equals true then          // local buffer is processed
4:              if message.lp is not set then                    // is broadcast
5:                  assignToLocalLPs(message)          // add LP assignments
6:                  forwardBroadcast(message)          // forward to remote sockets
7:              else
8:                  if message.lp is local then        // destination is on this socket
9:                      assignToLP(message)                      // add LP assignment
10:                 else
11:                     forwardToNode(message)  // forward message to target socket
12:                 end if
13:             end if
14:         else
15:             if message.lp is not set then                    // is broadcast
16:                 assignToLocalLPs(message)          // add LP assignments
17:             else
18:                 assignToLP(message)                          // add LP assignment
19:             end if
20:         end if
21:     end while
22: end function
```

ing partition would require as many outgoing buffers as living partitions present in the system. For that reason, we decided to employ a hierarchical approach for the message passing in ERIS as visualized in Figure 3.25. The central level of this hierarchy are single NUMA nodes that run a node coordinator that is responsible for managing the message passing within the lower level of the hierarchy (LPVs) and for passing messages on the global level (between NUMA nodes). The source of the message passing are micro operators that are executed by an LPV in the context of a living partition. At this level, the message passing layer tries to combine messages for space efficiency reason and the resulting messages are written to an LPV-local buffer. This local LPV message buffer may contains messages from multiple micro operators and is flushed periodically or when the buffer is full (1). The content of those buffers is written to the *Local Incoming Buffer*, which consists of two internal buffers. One internal buffer is currently written by LPVs and the other one is processed by the node coordinator, which is also executed by an LPV (cf., Figure 3.18). The internal buffers are periodically switched following the principle of a shadow buffer, which is implemented in the same way as the incoming buffers in our PoC (cf., Section 3.3.2). Complementary to the local incoming buffer, the ERIS message passing layer employs a *Remote Incoming Buffer* that operates similar to its local pendant. The difference between both buffers is that the local one is filled by local

LPVs, while the remote one is populated by node coordinators of the other sockets in the NUMA system (5).

Both incoming buffers (i.e., local and remote) are processed by the local node coordinator (2). The node coordinator either creates new LP assignments stored in the LP assignment queue (3) or forwards messages to other sockets. Forwarded messages are buffered in outgoing buffers (one for each remote socket) to reduce contention on the remote incoming buffers (4). Algorithm 3 describes the algorithm the node coordinator uses for processing either the local or the remote incoming buffer. The algorithm takes the respective buffer object as input and iterates over all messages (may consisting of multiple sub messages) of this buffer (line 2–21). If the buffer is the local incoming buffer and no target living partition pointer is set, the message is assigned to the local LPs of the data object (line 5) and is additionally forwarded to all nodes that contain LPs for the respective data object using the information of the data objects partition table (line 6). In case of a specified target LP, the message is either locally assigned (line 9) or forwarded to the target socket (line 11). If the buffer object is the remote incoming buffer, the message does not need to be forwarded to another socket, because messages in the remote incoming buffer already reached their respective destination node. This assumption is save for a static partitioning scheme without any *Data Placement Adaptivity*, which is not considered by this algorithm. Hence, the algorithm only distinguishes between a broadcast indicated by a `null` pointer and an unicast message. In the first case, the message is assigned to all local living partitions of the data object (line 16) and in the second case, the message is assigned to the respective target LP (line 18). The node coordinator always groups message assignments for the same LP during each run of the algorithm to increase the locality of the message processing.

To summarize, the message passing layer is – similar to the one employed in our PoC – built for high throughput to hide the latency of especially inter-socket communication, which can not be circumvented. This high throughput is achieved by a comprehensive buffering strategy on all levels of the message passing hierarchy, i.e., LPV level, intra-socket level, and inter-socket level. The buffering leverages the high sequential write performance of the main memory as well as the interconnects and additionally reduces the contention on buffers that are written by multiple threads in parallel.

3.4.5 Transactions

A central feature of database systems are transactions, which guarantee atomicity, consistency, isolation, and durability (ACID). Due to the additional costs that incur by ensuring all of those properties to a high degree, database systems define different isolation levels, each of them avoiding a certain set of anormalities (e.g., dirty reads or phantoms) that can occur when multiple transactions execute in parallel. Hence, an isolation level is a trade-off between performance respectively parallelism and the

degree of isolation. As the underlying implementation for concurrency control, nearly all modern DBMSs (e.g., SAP HANA [41], Microsoft SQL Server [39], Postgres [108], etc.) implement multi-version concurrency control (MVCC) [96, 140]. In contrast to traditional lock-based concurrency mechanisms, MVCC avoids in-place updates by maintaining multiple versions of modified tuples, which allows non-blocking reading transactions. In this section, we discuss how the MVCC mechanism is designed and implemented in ERIS to ensure the isolation level snapshot isolation (SI), which is mostly more than sufficient for most of the queries. Nevertheless, some application domains require serializable as isolation level, which is not guaranteed by the basic MVCC approach, but by certain MVCC extensions [100].

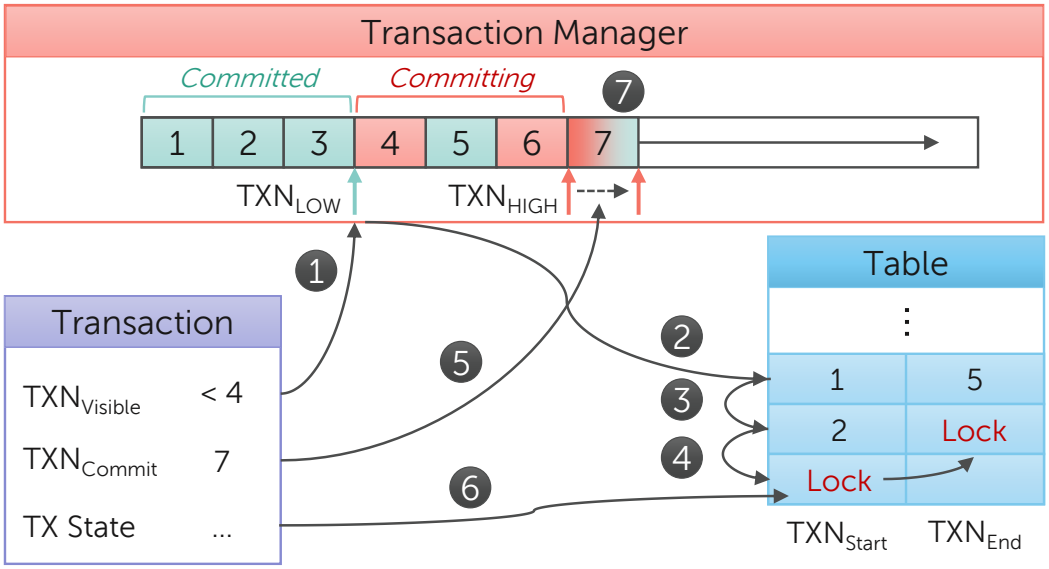


Figure 3.26: MVCC-based transaction processing example in ERIS.

In ERIS, transactions are started and committed respectively aborted by tasks and provide concurrency control for dataflows and their micro operators. Figure 3.26 visualizes the MVCC-based transaction concept with the help of an example where a transaction scans over a table and updates a single record in between. Once the transaction is started the data structure on the left hand side of the figure is instantiated, which mainly contains the $\text{TXN}_{Visible}$ time stamp, the TXN_{Commit} time stamp, and the transaction state (e.g., running, committing, aborted, or committed). The next step in the starting process (1) is to obtain the $\text{TXN}_{Visible}$ time stamp (a sequential number) from the transaction manager, which indicates the time stamp of the last successfully committed transaction that has no uncommitted predecessors. This time stamp is used to evaluate whether a record is visible for a transaction or not. As soon as the transaction obtained this time stamp, the transaction state is set to *running* and dataflows are ready to execute within the context of that transaction.

Afterwards, a micro operator iterates over the living partition of a table and faces its first tuple (2). Due to the MVCC concept, each table has hidden attributes that define the *start time stamp (TXN$_{Start}$)* and the *end time stamp (TXN$_{End}$)* of a record. For instance, the first tuple in the example was created by transaction 1 (start time stamp) and was deleted by transaction 5 (end time stamp). For our transaction, this version of a tuple is visible, because the start time stamp 1 is less than TXN$_{Visible}$ and the end time stamp 5 is greater than TXN$_{Visible}$. The second tuple version (3) is also visible for the transaction, because the start time stamp 2 is less than TXN$_{Visible}$ and the end time stamp is not set. The transaction updates this tuple by deleting this tuple version and creating a new version (4). To delete the tuple, a lock flag is set in the end time stamp and this lock is registered in the *Local Transaction Manager* of the respective living partition for two purposes: (1) To detect conflicts when another transaction tries to delete this tuple version, too. (2) To set the appropriate TXN$_{Commit}$ time stamp or to remove the lock when the transaction commits respectively aborts. The new tuple version is inserted as usual and sets a lock flag in the start time stamp instead of the end time stamp that is also registered in the local transaction manager for the same purposes. When the transaction commits (5), the TXN$_{Commit}$ time stamp is fetched from the transaction manager by incrementing the TXN$_{HIGH}$ number. Moreover, the transaction state is changed to *committing*. The next step in the commit process (6) is to notify all modified living partitions and ask their local transaction manager to apply the commit using TXN$_{Commit}$. This notification happens with the help of an assisting dataflow that sends commit messages to the LPs. Once the local transaction manager of an LP receives the commit message the locked start and end time stamps of the affected tuple versions are unlocked and become replaced by the TXN$_{Commit}$ time stamp 7. As soon as the commit dataflow finished, the transaction state is set to *committed* (7) and the transaction object is destroyed. The process of a transaction abortion is very similar except that TXN$_{Commit}$ is not even fetched and is thus, not set by the local transaction managers. If a transaction is read-only, the most costly steps 5 and especially 6 are not necessary, which makes read-only transactions lightweight. To remove tuple versions that are not in use anymore, a garbage collection is employed, which in integrated into our *Storage Adaptivity* approach (cf., Chapter 5).

To summarize, ERIS employs an MVCC-based mechanism for concurrency control that lets dataflows operate on an isolated snapshot of data objects. The transactional subsystem uses a hierarchical approach consisting of a global transaction manager that is responsible for managing the time stamps and living partition-local transaction managers that manage the start and end time stamps of their data object partition. While read-only transactions are pretty lightweight, writing transactions require additional communication with the global and local transaction managers.

3.4.6 Evaluation

In this section, we evaluate our living partitions-based DBMS research prototype ERIS in terms of scalability on large-scale NUMA systems. Compared to our previous PoC (cf., Section 3.3), ERIS provides a sophisticated infrastructure that supports transactions, generic tasks, dataflows, and a comprehensive storage layer that implements schema flexibility as well as a variety of physical storage formats to enable *Storage Adaptivity* (cf., Chapter 5). In the following, we describe our evaluation environment and present a set of microbenchmarks that cover the most important cases ERIS' scalability depends on. Finally, we will present and discuss the results of the TATP benchmark [62].

Evaluation Environment

The test hardware we are using for the evaluation of ERIS is a SGI UV 3000 system (cf., Table 3.1) consisting of 64 processors. Each processor is an Intel E5-4655 v3 (6 physical cores, 12 hardware threads, 30 MB LLC, 2.9 GHz) of the Haswell-EP generation equipped with 128 GB main memory resulting in a total memory size of 8 TB for the entire machine and an overall hardware thread count of 768. The test system has the same interconnect topology as the SGI UV 2000 investigated in Section 3.1 (cf., Figure 3.1(c)) and runs SUSE Linux Enterprise Server 12 as operating system.

We implemented ERIS in C++ supporting all of the features and concepts described in the previous section. Moreover, our implementation already uses an adaptive storage layer as it is required by our storage adaptivity concept. Nevertheless, we use a static physical data layout for all of our experiments in this section, but still face additional costs for the late binding of physical operators as we will show in Section 5. We hardcoded all of the tasks and queries that are necessary to run the experiments and precompiled them with the `g++` compiler similar to ERIS itself. Since ERIS deeply implements the living partitions architecture, we can not give a meaningful comparison to the transaction-oriented architecture. Hence, we use a version of ERIS that uses an interleaved memory allocation policy instead of a local one for comparison purposes and focus on the scalability of ERIS itself.

Microbenchmarks

The foundation for our microbenchmarks is a key-value store that uses 8 Bytes for the key and 8 Bytes for the value. The store uses a static range partitioning and equally distributes the key-value pairs across the living partitions. We create as much living partitions as living partition vitalizers are active on the system. Regarding the amount of key-value pairs that are generated, we use two scale factors (SF) with a base multiplier of $100,000$ pairs:

SF 1000. This scale factor generates 100 M key-value pairs amounting in a data size of 800 MB for the keys only. Since the effective cache size of our test system is

1.62 GB (64 times 30 MB cache per processor), this key set fits into the cache and thus, (1) memory locality is not the important factor and (2) the message passing layer is more stressed, because the data object access is relatively fast.

SF 5000. The scale factor generates 500 M key-value pairs resulting in a data size of 4 GB solely for the keys. This data size exceeds the effective cache size of our test system and hence, (1) local memory access is becoming an important factor and (2) the message passing layer faces a moderate stress, since data object accesses ar rather costly.

Table 3.4: Overview of the microbenchmark configurations.

Partitioning	Table Scan		Index Scan	
	SF 1000	**SF 5000**	**SF 1000**	**SF 5000**
By Key	Fig. 3.27	Fig. 3.28	Fig. 3.29	Fig. 3.30
By Value	Fig. 3.31	Fig. 3.32	Fig. 3.33	Fig. 3.34

For each microbenchmark configuration we execute an amount of tasks that is high enough to saturate the database system. A task continuously opens a transaction that executes 1,000 key-value lookups in parallel. Each key is unique and guaranteed to be present in the store. To systematically evaluate the scalability behavior of ERIS, we distinguish the cases shown in Table 3.4. First, we differentiate whether the key-value store is partitioned by the key attribute or not. In the first case, the message passing layer uses an unicast, because the target living partition is known by the routing table. In the second case, a broadcast to all living partitions of the store is necessary resulting in higher messaging and LP processing costs. The next difference we make is between a table scan and an index scan, because both access paths exhibit differences in terms of processing speed, cache usage, and memory access pattern, i.e., sequential or random access. In the following, we discuss all of the combinations in detail.

Table Scan by Key. In this scenario, a key lookup is resolved by sending a single message to the target LP that executes a column scan to find the corresponding value. The results for the first data set are presented on Figure 3.27. This chart shows the actual result (ERIS), the numbers for the interleaved version (ERIS Interleaved), the ideal linear scale-up (Linear), and the ideal quadratic scale-up (Quadratic). The ideal scale-up numbers are relative to the single-processor (12 logical cores) measurement of ERIS. This experiment shows that ERIS is able to scale better than the ideal quadratic scale-up in such a scenario, because (1) living partitions and hence, the amount of data that needs to be scanned is decreasing when employing more LPs, (2) the number of processing resources (LPVs) increases, and (3) the effective cache size increases when activating more processors and thus, more and more data fits into the cache. Moreover, we observe that the interleaved

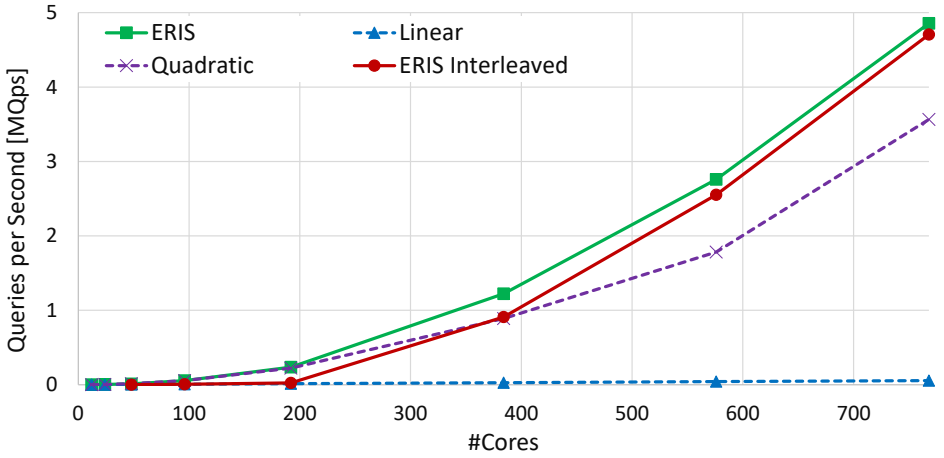


Figure 3.27: Scalability of the table scan by key in ERIS (SF 1000).

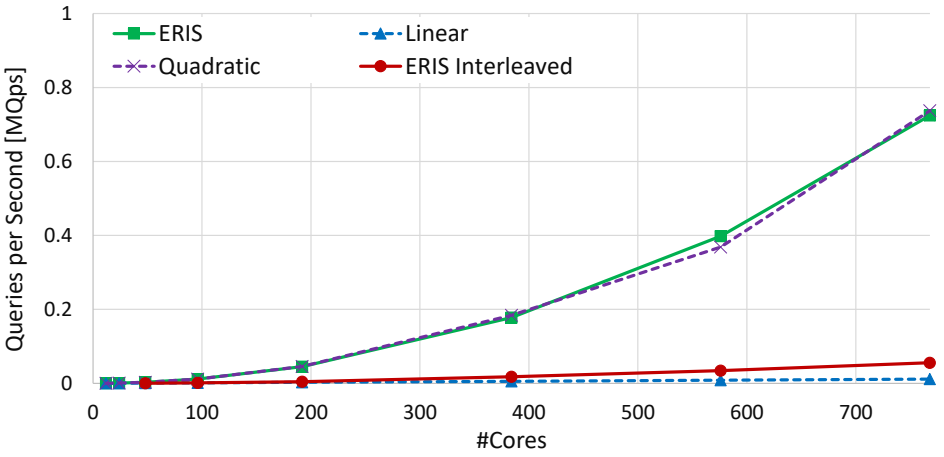


Figure 3.28: Scalability of the table scan by key in ERIS (SF 5000).

version of ERIS shows a similar behavior, because the data set is small enough to fit into the cache and hence, the memory access locality does not make a big difference. This scalability behavior of the interleaved ERIS version changes dramatically when looking at the numbers for the big data set in Figure 3.28. In this scenario, column scans are memory-bound and thus, the number of remote memory accesses become a severe scalability blocker. In contrast, ERIS is able to scale in the same way the ideal quadratic scale-up does, because statement (1) and (2) still apply.

Index Scan by Key. In this experiment series, we look at the scalability behavior of index scans using unicasts to a single target LP. This setup stresses the message passing layer, because index scans execute orders of magnitude faster compared to table scans respectively columns scans. Moreover, index scans are latency-bound and do not require a fast sequential memory access. In Figure 3.29, we show the

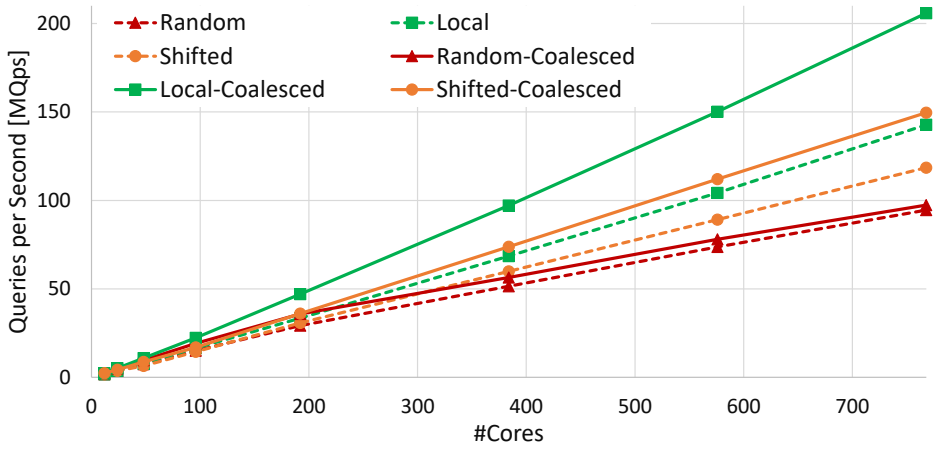


Figure 3.29: Scalability of the index scan by key in ERIS (SF 1000).

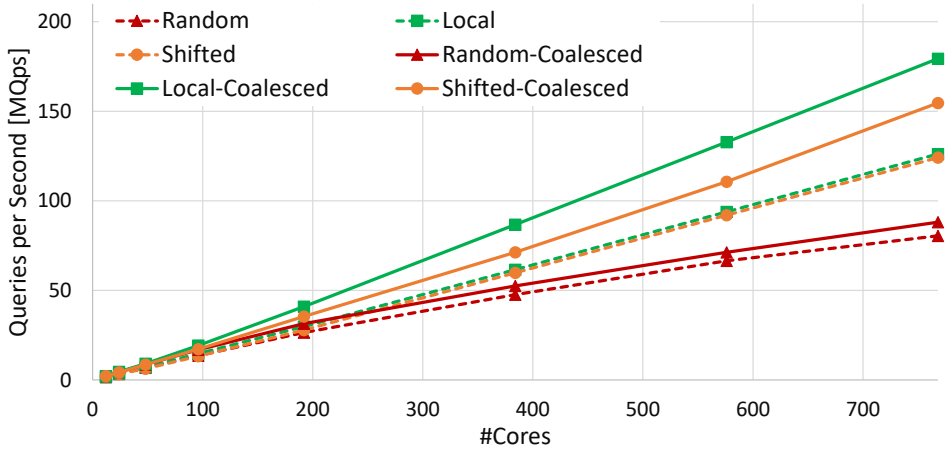


Figure 3.30: Scalability of the index scan by key in ERIS (SF 5000).

measurements for 6 different settings using the small dataset. In the first setting, tasks issue index scans randomly to all available living partitions (Random), while the second setting limits index scans to processor-local LPs only (Local). In the third setting (shifted), index scans are also issued to living partitions on a single processor, but this time the target processor is shifted by 32. For instance, all tasks on the first processor issue their index scans only to LPs on processor 32, which is a shift by half of the available processors (64 in total). All of the settings come in the standard flavor or in the coalesced flavor (X-Coalesced). In the standard flavor, LPs process incoming messages in a message by message way and in the coalesced flavor messages are processed in batches by living partitions, which results in a more efficient message processing, because of an optimized code path and an increased locality when accessing the data object.

In the random case, ERIS scales to 84% of the ideal linear scale-up for 768 cores, because of the high variety of target LPs, which is the worst case scenario for the message passing layer. Moreover, the message processing can not often be coalesced, because 768 target LPs for 1000 requests per task result in 1.3 messages per batch. However, we also observe a superlinear scale-up until 200 cores, because at this point we reach the equilibrium between the number of target LPs and the reduced size of partitions, which positively affects the cache-sensitive index data structures. In the shifted setting, ERIS increases its scalability to 106 % of the ideal linear scale-up. Since all requests of a task are issued to living partitions on the same processor, the message passing layer can efficiently transfer large message batches between processors, which results in a lot of positive side effects (e.g., lower contention on remote incoming buffers and data locality during node coordinator processing). Regarding the coalesced flavor of the shifted setting, we observe an improvement of 20 % for the peak performance compared to the non-coalesced flavor, because 12 target LPs per 1000 requests result in 83.3 requests per batch. Nevertheless, the full batch size is rarely reached, because messages are split into smaller pieces in the stages of our hierarchical message passing layer. Finally, we see the best scalability behavior in the local case, which amounts to 127 % of the ideal linear scale-up and is thus superlinear similar to the shifted case. Furthermore, the coalesced flavor reaches 30 % improvement, which is 10 % more compared to the shifted case, because message batch splits are less probable, since only the local level of the message passing layer hierarchy in involved.

In Figure 3.30, we show the measurements for the big dataset, which effectively causes more memory accesses per index scan and thus, increases the time need per key lookup. Compared to the small dataset, we observe lower absolute throughput numbers as well as changes in the relative numbers regarding the relation to the ideal linear scale-up (Random: 78 %; Shifted: 119 %, Local: 120 %). Except for the shifted case, these relative numbers are slightly lower compared to the small dataset caused by the increased cache traffic, which also affects the message passing layer. If we compare the absolute numbers to our PoC (cf., Figure 3.16), we observe an about 35x difference for index scans, because ERIS is implemented in a highly generic way and supports more features (e.g., transactions, record management, comprehensive message format, and a generic storage layout) compared to our highly specific PoC. Contrary, the performance of table scans is similar for the full ERIS implementation and our PoC, because orders of magnitude less data object accesses are issued and the scan itself faces not much additional overhead.

Table Scan by Value. In this scenario, we investigate the scalability behavior of ERIS in the context of table scans when the table is not partitioned by the target attribute (key). In such a case, the message passing layer needs to generate broadcasts that triggers all living partitions of the table to scan their respective partition. Once again, all of the key-value pairs are equally distributed across the individual LPs. We show the results of this scenario for the small dataset in Figure 3.31. As shown, ERIS scales superlinearly, because (1) the partitioning is getting more and

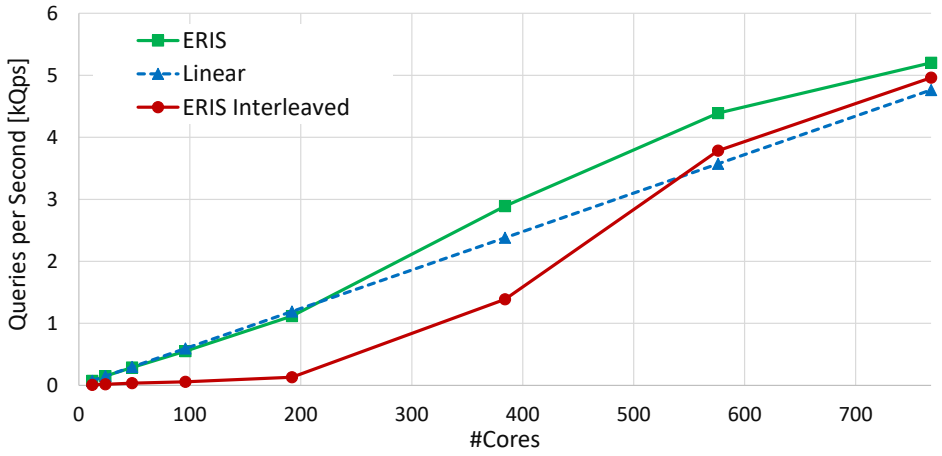


Figure 3.31: Scalability of the table scan by value in ERIS (SF 1000).

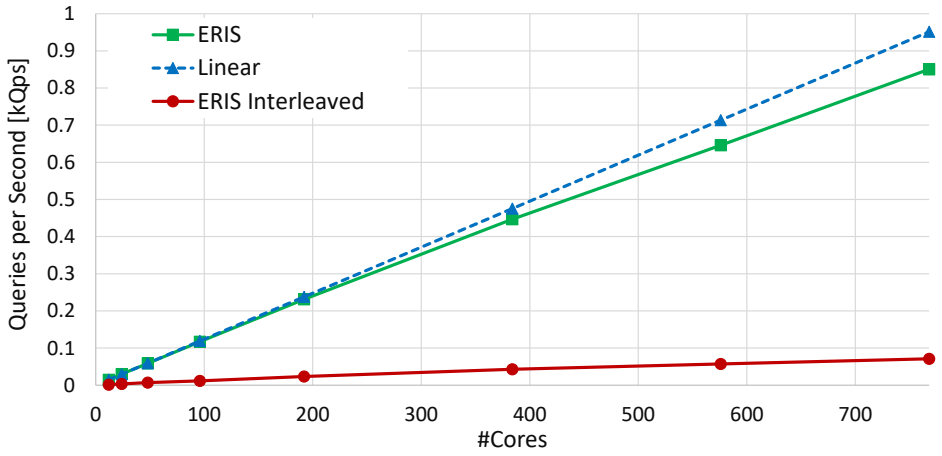


Figure 3.32: Scalability of the table scan by value in ERIS (SF 5000).

more fine-grained, which allows to activate additional compute resources, and (2) the effective cache size increases due to the activation of more processors. Compared to the table scan with unicasts (cf., Figure 3.27), all LPs have to scan their partition for a single request in the broadcast setup and thus, no direct advantage is generated from the increasing partitioning granularity. The figure also shows the measurements for the ERIS version, which uses an interleaved memory allocation policy. As shown, this ERIS version scales worse until 200 cores and is afterwards getting closer to the standard ERIS version that uses a local memory allocation policy. The reason for this behavior is that the effective cache size increases, because more processors are activated, and the dataset starts to fit into the cache, which eliminates the need for memory accesses.

Figure 3.32 shows the results for the big dataset. The difference compared to the small dataset is that the table scans are now memory-bound even when all processors are used. As the measurements show, ERIS scales slightly sublinear in this case, because the scan can only profit from the additional processors and thus, memory controllers, that are activated without any additional benefit from the increased cache size. Moreover, for each key lookup a broadcast is generated, which needs to be distributed across all processors including a processor-local distribution to all affected LPs, which induces an additional overhead. The interleaved ERIS version scales worse for all of the time, because memory accesses are the dominant factor in this setup.

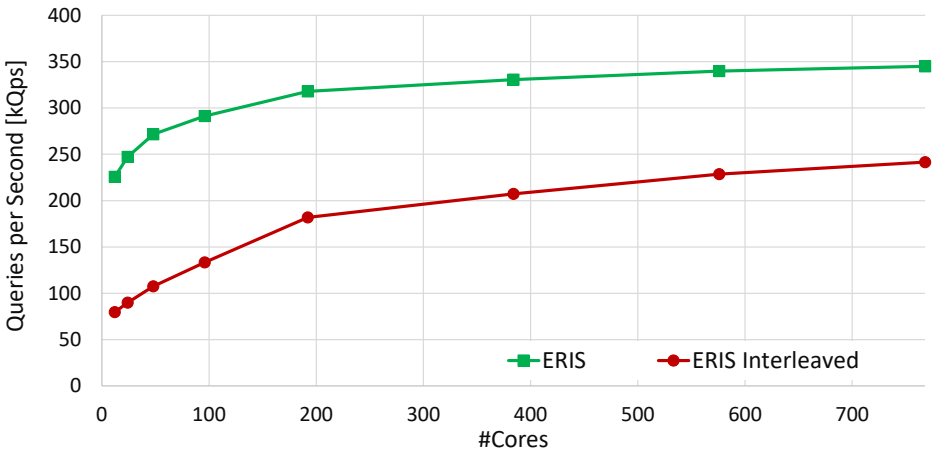


Figure 3.33: Scalability of the index scan by value in ERIS (SF 1000).

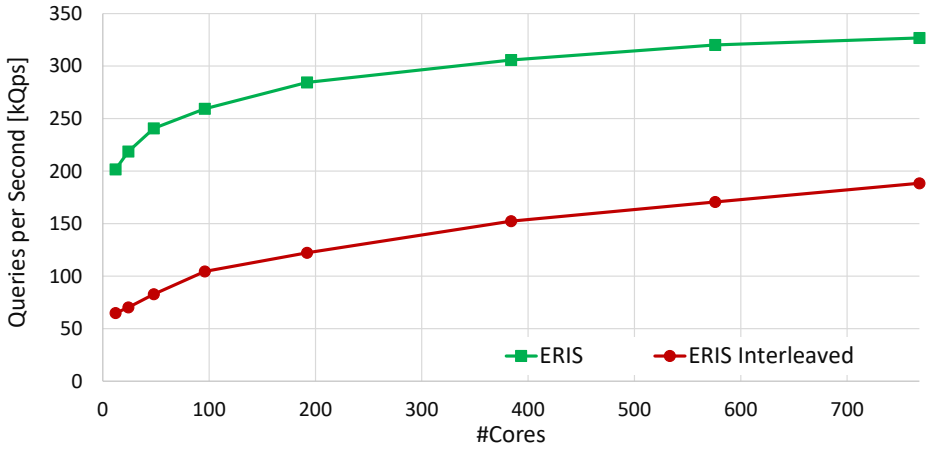


Figure 3.34: Scalability of a the index scan by value in ERIS (SF 5000).

Index Scan by Value. In the last scenario, we look at the scalability behavior of ERIS for index scans using broadcasts. Hence, the difference to the previous scenario is that operations execute much faster and are memory latency-bound. Nevertheless, ERIS still faces the issue that broadcasts generate a high amount of traffic in the message passing layer, because an index scan needs to be triggered for each and every living partition of the table when looking up a single key. This scenario is likely to happen, because a relation can only be partitioned by a specific set of attributes at once. We visualized the measurements for the small dataset in Figure 3.33. Unlike the previous scenarios, this specific setup scales only for a little amount of compute resources, because the only advantage we benefit from is that a higher portion of the index data structure fits into the cache when activating more processors. However, this effect is limited, because of the logarithmic complexity of tree-based data structures. The version of ERIS that uses an interleaved memory allocation policy scales similarly, but with lower absolute numbers, because the last levels of the index require main memory accesses.

In Figure 3.34, we show the results for the big dataset. The abstract shape of the chart looks nearly the same as for the small dataset. Nevertheless, we (1) observe a slightly improved scalability and (2) a bigger gap between the standard ERIS version and the interleaved ERIS version. Both differences are a direct consequence of the increased index data structure size and the portion of it that fits into the cache. Nevertheless, the conclusion of the evaluation of this scenario is that it is not feasible to employ a large number of living partitions when doing index accesses via broadcasts, because of the limited scalability. To cope with this issue, we suggest two approaches. The first way is to reduce the amount of LPs with the help of *Data Placement Adaptivity*, which includes the merge process that effectively does this job. The other way is to enhance the message passing layer with bloom filters [19] to reduce the number of LPs that receive the broadcast. However, the bloom filter approach faces high maintenance costs in the presence of data placement adaptivity, which is not in the scope of this thesis. Hence, we will leave this topic open for future work.

TATP Benchmark

The Telecommunication Application Transaction Processing Benchmark (TATP or TM1) is designed to measure the performance of a typical telecommunications application. The benchmark was modeled after a real test program that was used by a telecom equipment manufacturer to evaluate the applicability of various relational database systems to service control programming in mobile networks [62]. TATP consists of four tables (Figure 3.35) that are arranged as a snowflake schema and simulates a transactional workload (OLTP) on the database system by issuing a predefined set of transactions that include combinations of point selects, range selects, joins, inserts, deletes, and updates. The TATP-Mix specifies the combination of such transactions.

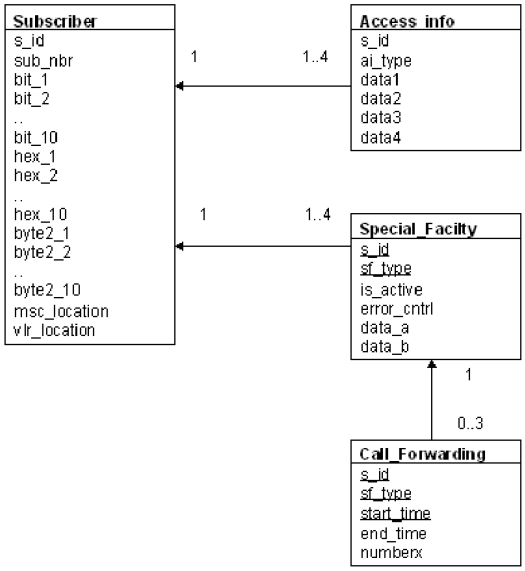


Figure 3.35: TATP database schema [62].

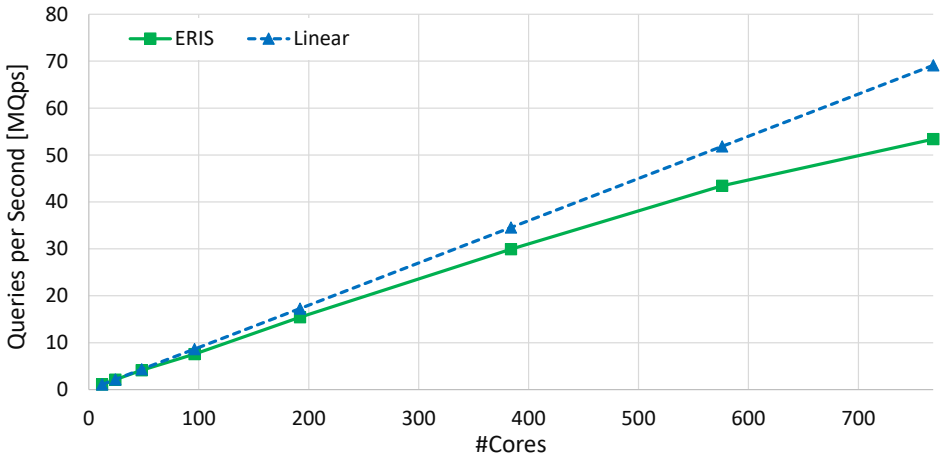


Figure 3.36: Scalability of the TATP-Mix in ERIS (SF 10).

In Figure 3.36, we give the scalability results for the TATP-Mix on ERIS using a scale factor of 10. We partitioned all tables by their respective primary key and equally distributed the data across the living partitions. Note that *Data Placement Adaptivity* is out of scope for this thesis and thus, the partitioning is static. Moreover, all beneficial indexes were present and we used the equal distribution random number generator. The chart includes the actual measurements as well as the ideal linear scale-up relative to the single-processor performance. As shown, ERIS is able to scale up to 77.2 % of the ideal linear scale-up. Since, all feasible indexes were created, most of the reading accesses use an index scan via an unicast (cf., Figure 3.29 and 3.30).

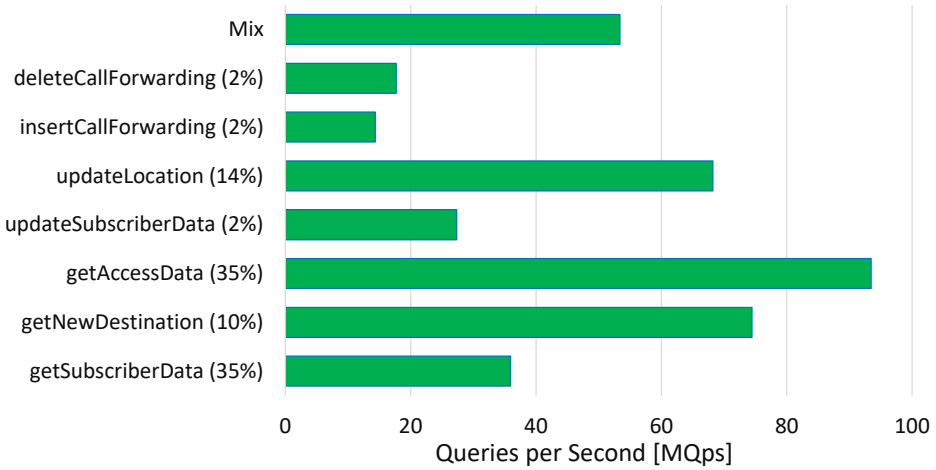


Figure 3.37: Throughput of the individual TATP transactions in ERIS (SF 10).

In Figure 3.37, we depict the breakdown of the TATP-Mix and show the peak performance numbers of the individual transaction types as well as their probability to be executed, which is defined by the benchmark itself. The `getSubscriberData` and the `getAccessData` transaction types are basic point accesses. We observe a huge performance difference between both transaction types, because the `getSubscriberData` transaction includes 34 attributes in the projection clause, while the `getAccessData` transaction includes only 4. Thus, the storage layer is responsible for that performance difference, because a high amount of attributes needs to be extracted from each tuple. The `getNewDestination` transaction type is a join between two tables that issues two highly selective successive `UnorderedScans`. As the throughput measurements show, ERIS is able to execute such transactions efficiently compared to the `getAccessData` throughput. The `updateLocation` as well as the `updateSubscriberData` transaction types issue point accesses and update the matched record. In ERIS, updates are handled by triggering an `UnorderedScan`, which deletes the matched record by setting the respective end timestamp. At the same time, an `Insert` message is sent to the new target LP that includes the new tuple. Additionally, such manipulative transactions need to send `Commit` messages to the modified living partitions as soon as the transaction commits. The `UpdateLocation` transaction type does only a single update, which results in a moderately lower throughput compared to the read-only `getAccessData` transaction type. In contrast, the `updateSubscriberData` transaction type executes two updates sequentially resulting in a slightly less than half of the throughput of the `UpdateLocation` transaction type, because both updates are executed one after another according to the definition of a transaction. Finally, the least performing transaction type `insertCallForwarding` and `deleteCallForwarding` do a series of select and insert respectively delete queries that require multiple dataflows and task interruptions.

3.4.7 Summary and Conclusions

In this section, we presented our data management system *ERIS*, which is designed from scratch to implement the *Living Partitions* architecture that extends the data-oriented architecture to fulfill all of our requirements for an energy-aware database system and thus, mainly enables scalability and fine-grained adaptivity. We introduced the overall architecture of ERIS, which uses *Living Partition Vitalizers (LPV)* as the main vehicles for spending actual compute resources for various roles such as *Tasks*, the *Living Partitions* itself, and the *Node Coordinator*. Each LPV is statically pinned to an exclusive hardware thread of the system.

Tasks are one main component of ERIS and represent the sequential portion of one or multiple transactions. They are written in C++ leveraging the *ERIS/C++ Framework* to control transactions as well as to construct and control dataflows. To actually process and modify data objects, *ERIS* employs *Dataflows* that are executed in the context of a transaction and consist of *Micro Operators* that communicate via messages. Hence, we introduced the message passing layer of ERIS, which – compared to our PoC – supports comprehensive queries, transactional properties, and the living partitions architecture. The message passing layer is organized hierarchically in an intra-socket level and the inter-socket level. Both levels are mainly controlled by the special *Node Coordinator* role that exists once per socket and is also executed by an LPV. To support transactions as a central feature of database systems, ERIS uses an multi-version concurrency control (MVCC) based approach that is organized in a local transaction manager per living partition and a single global transaction manager.

We evaluated ERIS' scalability behavior using a systematic set of microbenchmarks to investigate the most important scenarios that can occur comprising combinations of unicasts and broadcasts as well as table scans and index scans. The micro benchmarks showed that ERIS scales linear or even superlinear in most of the scenarios and that local memory allocation is an important factor for the overall scalability. Nevertheless, we also identified a scenario where the scalability is limited to a small number of resources and suggested appropriate countermeasures. For the end-to-end evaluation we used the TATP benchmark and demonstrated that ERIS also scales up in real-life OLTP situations.

3.5 Summary and Conclusions

In this chapter, we started with an exploration of current medium and large scale-up NUMA system architectures to quantify and asses the impact of remote main memory accesses on such architectures. Especially on large-scale NUMA systems, our experiments revealed that latency and throughput differ up to an order of magnitude when accessing main memory remotely, which emphasized that local main memory access is the key factor for scalability on such hardware platforms. Based on those insights, we classified existing DBMS architectures in terms of their ability to scale-up on large NUMA systems as well as their ability to allow fine-grained adaptations at runtime. We concluded that – compared to the transaction-oriented architecture – the data-oriented architecture provides us with the best foundation for fulfilling our requirements for an energy-aware DBMS. Nevertheless, this architecture still lacks (1) an investigation and appropriate concepts for in-memory DBMSs on large-scale NUMA systems as well as (2) certain requirements originating from our adaptivity facilities.

To cope with the first issue, we transferred existing concepts of the data-oriented architecture from medium-scale disk-based systems to large-scale in-memory systems, which is mainly a matter of the message passing subsystem that needs to keep pace with the increased speed of data object accesses. We implemented the corresponding proof of concept (PoC) to evaluate our concepts and focused on database primitives such as scans and index accesses. Our evaluation showed that the data-oriented architecture is able to scale up on large-scale NUMA systems in the context of an in-memory database system and clearly outperforms the classic transaction-oriented architecture. Our in-depth evaluation also reflected on the root causes for this scalability gap between both architectures. Moreover, we demonstrated that *Data Placement Adaptivity* can efficiently be done in such an environment.

To address the second issue, we extended the data-oriented architecture to enable fine-grained adaptivity at runtime. Hence, we presented the *Living Partitions* architecture, which enables a flexible work to hardware thread assignment as well as a late-binding of physical operators. Afterwards, we introduced our in-memory data management system *ERIS*, which was designed from scratch to implement the living partitions architecture as well as our *Adaptivity Facilities*. In contrast to our PoC, ERIS is able to execute comprehensive queries in a transactional environment using constructs like *Tasks*, *Dataflows*, and *Micro Operators*. Furthermore, ERIS employs a hierarchical message passing layer, to deal with the changes introduced by the living partitions architecture. In the evaluation of ERIS, we used a systematic series of microbenchmarks as well as the standardized transactional TATP benchmark and demonstrated the superior scalability of ERIS on large-scale NUMA systems. Hence, ERIS and its living partitions architecture are an excellent foundation for investigating our adaptivity facilities with the overall goal to build an energy-aware data management system.

4 Resource Adaptivity

To improve both metrics for energy awareness, namely *energy efficiency* and *energy proportionality*, modern processors provide a rich set of knobs for controlling power and performance at runtime. These knobs are independent core and package sleep states (C-states), individual core and uncore clocks (P-states), HyperThreading, energy-efficient turbo modes (EET), as well as per-core EPB (Energy-Performance Bias) settings. Due to the limited energy awareness of applications, which is mostly completely absent, current CPUs and operating systems try to manage most of these power management facilities on their own, but are limited in their energy saving potential, because of limited application-specific knowledge. Especially for state-of-the-art in-memory database systems, which manage most of their resources on their own, CPU and operating system have only little chance for efficiently tuning the server power consumption. Moreover, in-memory database systems are an application class that makes heavy use of the main power consumers of a server, which are CPU cores and main memory, and have to meet certain query latency targets making them an attractive subject for energy tuning.

So far, energy-related and hardware-centric research in the context of scale-up database systems concentrated on the energy consumption analysis of database servers [134], on finding the best core frequency setting for certain queries or operators [44, 57, 109, 143], or adding energy efficiency as an additional target to the query optimizer [83, 144]. These techniques allow the database system to make a binary decision between executing queries in performance or power-saving mode. Those methods are a good starting point, but do not consider target query latencies, ad-hoc queries and inter-query effects. Other recent approaches use a feedback-control loop to accomplish throughput or latency goals [81, 135, 145]. However, these works mainly address fairly parallel disk-based database systems and most of the energy savings originate from the slow disk accesses whereas highly parallel in-memory DBMSs mainly face bottlenecks within processors and memory controllers.

In this chapter, we present *Resource Adaptivity* as a hardware-centric *Adaptivity Facility* that follows our core concept of *Energy Awareness by Adaptivity*. Resource adaptivity is a holistic approach for adaptive energy-control of scale-up in-memory database systems that is able to deal with arbitrary short-running queries and saves energy while trying to stay within a user-defined query latency constraint. As depicted in Figure 4.1, our approach implements the hierarchically organized *Energy-Control Loop (ECL)* (cf., Section 2.5.2) that uses a socket-level energy profile, which is continuously maintained at runtime to adapt to changing workload characteristics. The *ECL* monitors the current system state and is frequently adapting the compute resources. Doing so, the *ECL* is able find the most energy-efficient degree of paral-

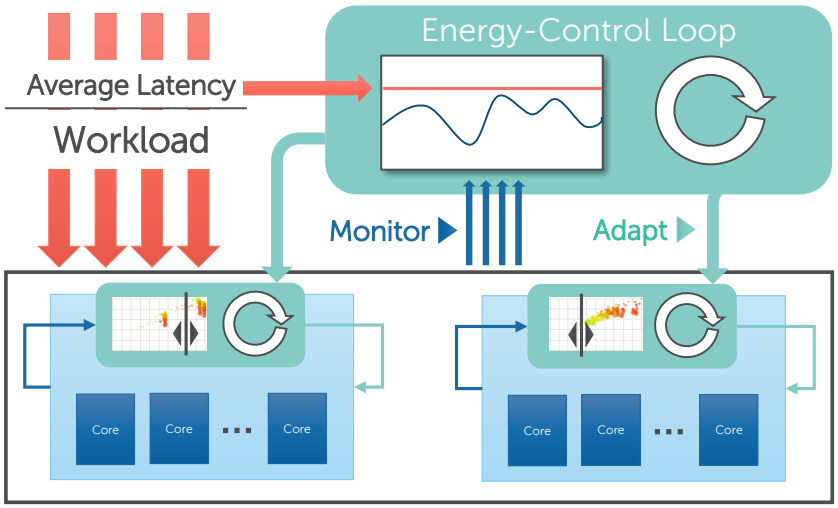


Figure 4.1: Resource adaptivity-specific energy-control loop.

lelism as well as core and uncore frequency settings for the current workload type and system load. Moreover, the *ECL* is able to discover higher performing resource configurations compared to the traditional all-in strategy.

We start with a discussion of the related work followed by an analysis of the energy tuning knobs of a current mainstream server system and quantify the impact of the respective knobs on performance and power. Afterwards, we propose energy profiles and describe how they can efficiently be generated and maintained at runtime. With the help of such profiles, we show how different workload types affect the optimal compute resource configuration in terms of energy efficiency and performance. Finally, we propose the resource adaptivity-specific energy-control loop as a holistic approach for adaptive energy-control on scale-up in-memory DBMSs. Contrary to existing approaches, the ECL works for arbitrary short-running queries and saves energy by frequently adapting the compute resource configuration while trying to stay within a given query latency constraint. We will give an exhaustive evaluation of our resource adaptivity approach regarding all important aspects including an end-to-end evaluation using a real world load profile.

4.1 Related Work

The energy efficiency of database servers is a critical research topic that requires action from the hardware as well as the software side [57, 105], because the scalability of database servers is limited by the "energy wall". From the software side, research so far focused on the energy analysis of database systems and on the active usage of energy-control knobs offered by hardware. Other approaches suggest changes in hardware to cope with the issue of energy efficiency. In the following, we will detail on the individual related works in the respective area.

4.1.1 Energy Analysis

Early works started with analyzing the potential of DVFS to increase the energy efficiency of a database server [134]. The main conclusion was that this approach is not feasible, because of the high static power consumption of the hardware that was available at that time and thus the authors concluded that the most energy-efficient configuration is the most performing and power consuming one and that energy optimizations are only feasible at cluster level. As we will show in Section 4.2, these findings do not hold anymore for current servers. Other studies [44, 143] found that software optimizations (i.e., the query optimizer) can significantly improve energy efficiency of a DBMS by considering energy as an additional optimization goal. While the previous works did their analysis at query-level, Psaroudakis et al. [109] analyzed the energy consumption of individual database operators and concluded that fine-grained scheduling mechanisms are able to further improve the DBMS energy efficiency. However, all of these works did their energy analysis either on outdated hardware or did not consider the additional energy-control knob offered by a current system such as independent core and uncore frequencies as well as the energy-performance bias (EPB). Within our work, we provided an exhaustive analysis of a current mainstream server and quantified the effects of all available energy-control facilities for a variety of workload types and additionally considered NUMA-related effects.

4.1.2 Active Energy-Control

In the context of distributed database systems, several approaches [84, 93, 118] tried to achieve energy proportionality by dynamically powering individual servers down or up. Because of the high costs for moving data and power cycling single servers, those approaches are only applicable as long term solutions and negatively affect energy efficiency, since data movement consumes a high amount of energy and scale-up architectures usually exhibit a better performance compared to scale-out solutions. Within a single database server, some works [83, 143, 144] dealt with adding energy as an additional optimization criteria to the query optimizer of the MySQL or Postgres DBMS. To do so, the authors built and calibrated energy and performance models for single plan operators that are leveraged by the optimizer. The energy savings amounted to about 20 %, but did not consider a response time limit and the evaluation was limited to a dual-core system, since MySQL and Postgres are disk-based DBMS with limited intra-query parallelism. Another class of active energy-control approaches uses a feedback control loop to dynamically adjust DVFS settings at runtime. For instance, Tu et al. presented the Postgres extension E^{2}DBMS [135] that adaptively controls the DVFS setting (one per processor) and the power state of the hard disks, while obeying a query throughput target. The authors assume a monotonic relationship between power and performance, which is mostly not the case as we demonstrated in Section 4.3.2. Another approach [145] relies on a workload classifier that uses the amount of disk I/O to select the appropri-

ate DVFS setting, since a high disk I/O rate makes a lower CPU power mode more energy-efficient. The recent LAPS [81] approach employs a feed-forward control mechanism that tries to stay within a certain response time limit. LAPS operates at the core level and adjusts the DVFS setting per core and is able to cope with short-term workload fluctuations. Additionally, the authors propose memory sizing as an additional tool for energy control. However, the main drawback of LAPS is that a precise query execution time model is needed, which is hardware and DBMS dependent and should consider other queries that are running in parallel. Moreover, this approach is not suitable for intra-query parallelism. Compared to our *ECL*, the mentioned works mainly address fairly parallel disk-based database systems where disk accesses are the main bottleneck. Contrary, the *ECL* primarily focuses on in-memory DBMSs that run on massively parallel scale-up hardware and bottlenecks usually occur within processors and memory controllers.

4.1.3 Hardware Solutions

A completely different way of achieving energy efficiency is building specialized hardware [80, 142], like FPGAs or ASICs, or to augment the existing instruction set architecture (ISA) by database-specific instructions [12]. Those approaches are able to save energy in orders of magnitude, but face high costs for hardware development. Another promising direction are asymmetric multiprocessor systems (AMP) like the ARM big.LITTLE architecture, which lets energy-efficient low performance cores coexist with power-hungry high performance cores inside of a single server. Such heterogeneous hardware setups are an interesting research direction for our *ECL* and could help to improve energy efficiency in low performance ranges [138].

4.2 Energy-Control in Current Mainstream Servers

In this section, we analyze performance and power characteristics of current mainstream server systems. Especially, we focus on the available energy-control features and the energy-related decisions that are made by the CPU itself. First, we will specify our test system and discuss the available options for measuring the power consumption. Afterwards, we will present our experiments regarding the static and dynamic power consumption of such a system, the impact of C- and P-States, as well as the decisions made by the EPB.

4.2.1 System Under Test

Due to the near-total market dominance of Intel, we are investigating a representative 2-socket sever-class system equipped with Intel Xeon E5-2690 v3 CPUs (Q3 2014) of the Haswell-EP generation and 256 GB DDR4 RAM (8x 32 GB PC4-2133 LRDIMMs)[1] (cf., Table 3.1). This system is similar to a single blade of the SGI

[1] Due to administrative reasons we were not able to conduct our experiments on the SGI UV 3000 system

UV 3000 system, except that the processors have twice as much hardware threads than the large-scale SGI system. Each CPU consists of 12 physical cores resulting in 24 hardware threads with HyperThreading enabled. Compared to previous generations, the Haswell-EP generation includes a portfolio of new energy-control features. One feature is the use of fully integrated voltage regulators (FIVR).

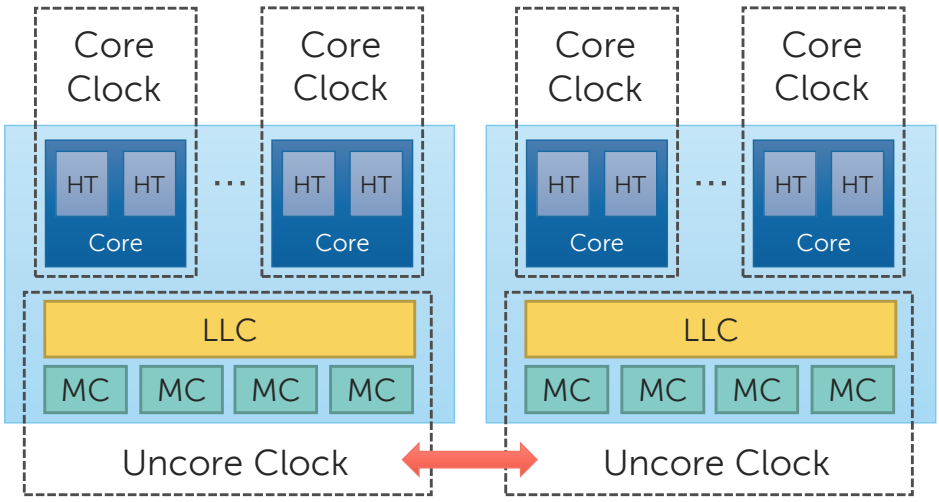


Figure 4.2: Available clocks on Haswell-EP CPUs.

While previous generations use a single clock for the whole processor, FIVRs enable independent clocks for individual parts of the CPU that allow a more fine-grained dynamic voltage and frequency scaling (DVFS). Figure 4.2 shows the available clocks inside our evaluation system. Each physical core has a separate clock that is shared by its hardware threads. Moreover, each processor features a separate uncore clock, that affects power and performance of the last-level cache (LLC) and the four memory controllers. Additionally, the CPU implements features such as the energy-efficient turbo (EET) and the energy-performance bias (EPB), which we will investigate in the remainder of this section. To measure the energy consumption of the system we either use an LMG450 power meter that is attached to the power supply unit, or the per-processor integrated RAPL counters (cf., Section 2.4.1). RAPL counters are highly accurate on this platform and allow us to separately measure the power consumption of the package domain (cores and caches) and the memory controller domain.

4.2.2 Static and Dynamic Power Consumption

With our first experiment, we figure out which components of the system draw which amount of power in idle mode and under full load. To do so, we first measure the power values from the external power meter as well as the internal RAPL counters in idle mode and once again under full load. To get the system under full load,

we use the FIRESTARTER tool [49] that uses the optimal balance of compute instructions, AVX instructions, and memory controller requests. The results are visualized in Figure 4.3. Note that the figure does not include the turbo peak of 500 Watts, since this high load can only endure for about 1s due to thermal limitations. The main conclusions we can draw from this experiment is that the static power consumption of the server system is only about 18 % of the peak power, which is a great advancement compared to the number of over 50 % reported in 2010 [134] and thus, opens up a lot of space for energy optimizations. Moreover, we can see that the largest amount of the dynamic power is consumed by CPU and DRAM, which also generates a dynamic power overhead – originating from power conversion losses and CPU fans – of about 15 % that can not be measured by RAPL counters. However, since attaching power meters to servers is not a practical solution, we will stick with the RAPL measurements, which are proven to accurately correlate with the PSU power consumption [50].

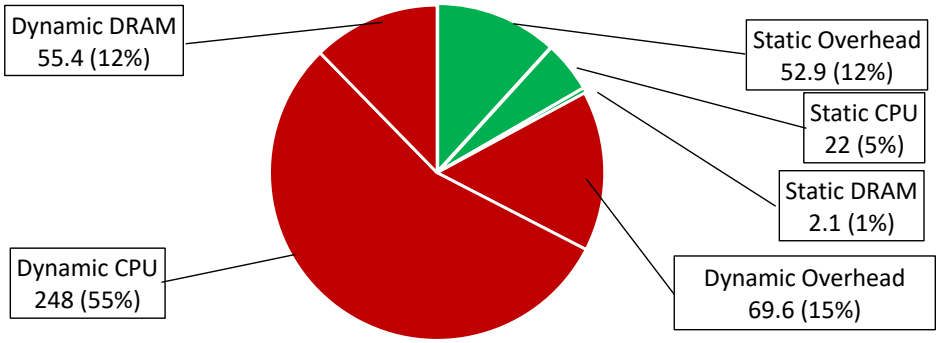


Figure 4.3: Haswell-EP power breakdown into static and dynamic consumers.

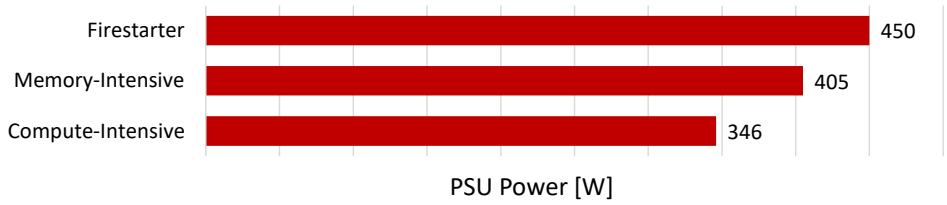


Figure 4.4: Haswell-EP maximum power consumption for different workloads.

In an additional experiment, we want to measure the impact of the workload type on the maximum power consumption of the test system. We use three workload types, which use all of the available hardware threads on the machine. (1) The aforementioned FIRESTARTER tool. (2) A memory-intensive workload that simulates a columns scan. (3) A compute-intensive workload that operates on cache-resident data. In Figure 4.4, we show the respective maximum power draw measured via the external power meter for the different workload types. The least amount of power

is drawn by the compute-intensive workload, because only the cores themselves are activated. In contrast, the memory-intensive workload additionally activates the memory controllers of both processors, which adds an additional power consumption of about 60 W. We measured the highest power draw with the FIRESTARTER workload, because the additional usage of AVX instructions increases the actual power consumption even more.

4.2.3 C-States and P-States

On modern CPUs, single cores or the entire processor can be power-gated to save energy, if they are not utilized (C-states). Additionally, the hardware implements power states (P-states), which decrease voltage and frequency to operate in a more energy-efficient state at the cost of reduced performance. One innovation of the Haswell-EP generation is that the individual core clocks as well as the uncore clock can be set independently (cf., Figure 4.2). On our test system, core clocks can be set between 1.2 and 2.6 GHz (3.1GHz TurboBoost) and the uncore clock ranges from 1.2 GHz to 3.0 GHz.

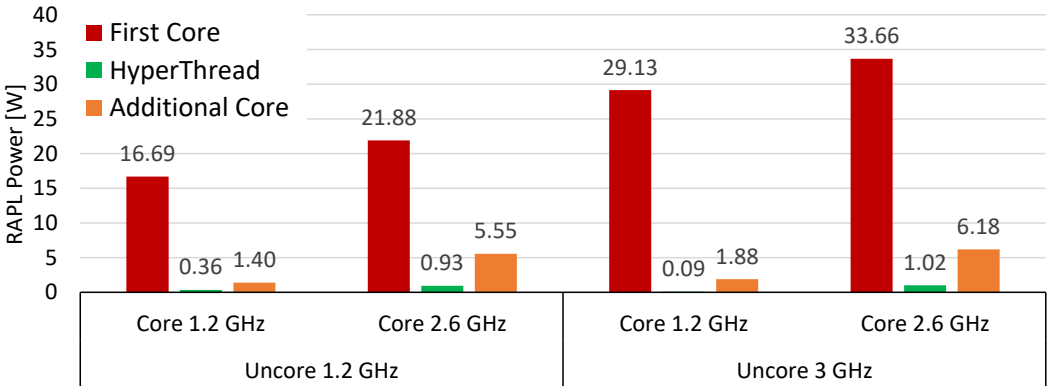


Figure 4.5: Power costs for activating cores with different core and uncore frequency settings.

In Figure 4.5, we experimentally evaluated the impact of C-states and P-states on the power consumption for a compute-intensive workload (no main memory accesses to avoid main memory-related bottlenecks) using RAPL counters on a single socket. The results show that most of the power costs incur when the first core of a socket is activated, while activating an additional core causes a much lower power draw and activating HyperThread siblings comes at almost no cost. The high power costs for activating the first core adhere to the uncore clock, which can be halted, if no core is active on the CPU. Thus, we observe a correlation to the uncore frequency. Halting the uncore clock allows the processor to power-gate the power-hungry LLC, which saves up to 30 W. The power costs for activating additional physical cores are almost constant, which can be considered as an energy-proportional process, and different core frequencies can be mixed-up, due to the independent core clocks.

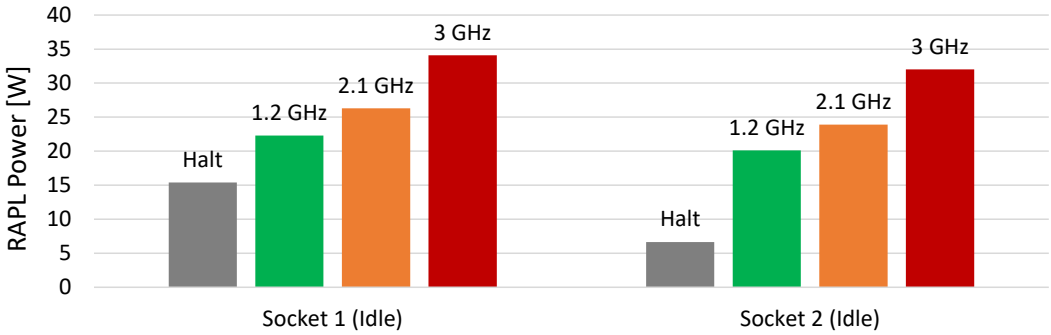


Figure 4.6: Socket-specific power consumption and cross-socket dependencies.

However, halting the uncore clock in a multi-processor system depends on the uncore state of the other processors in the server. Since CPUs are able to access memory on a foreign socket, the uncore clock of a socket can not be halted unless all processors of the system halted their uncore clock, too. Figure 4.6 shows the power consumption of the individual sockets for a halted uncore clock (both sockets idle) and for different uncore frequencies, when the other socket is active. The experiment shows the mentioned dependency and also demonstrates that the second socket is consuming less power compared to the first one, especially when the uncore clock of all processors is halted.

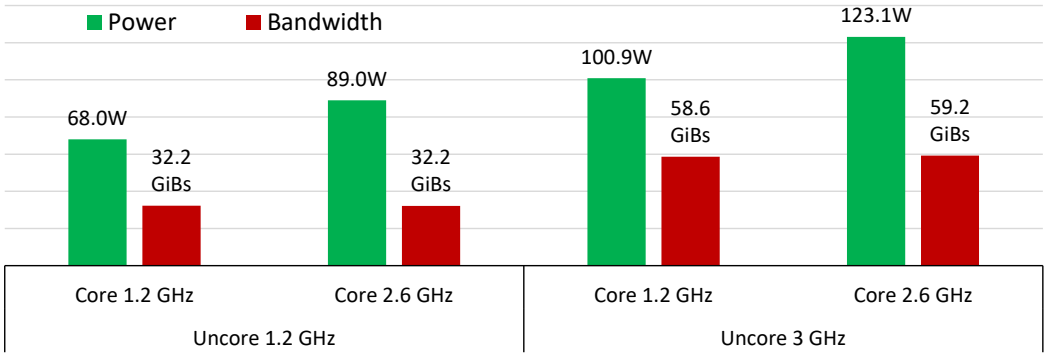


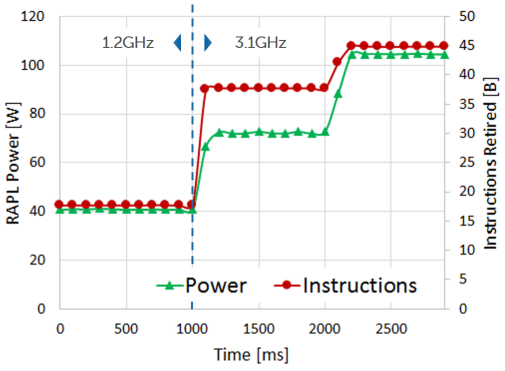
Figure 4.7: Memory bandwidth and power costs for different core and uncore frequency settings (all cores are active).

In our last experiment, we want to quantify the impact of core and uncore frequenices on the memory bandwidth. As Figure 4.2 suggests, the memory bandwidth heavily depends on the uncore frequency, because it affects memory controllers and the LLC. Our experiment in Figure 4.7 confirms this assumption. It clearly shows, that the available memory bandwidth mostly depends on the uncore frequency setting and that nearly the full bandwidth can be achieved when operating all of the

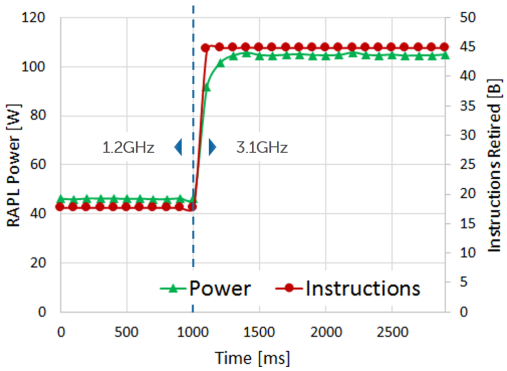
cores in their lowest available P-state (1.2 GHz) as long as the uncore clock is set to its respective maximum (3.0 GHz).

4.2.4 EPB-Driven Energy Management

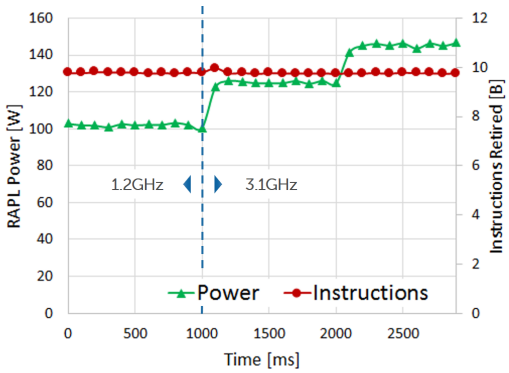
The energy-performance bias (EPB) can be set per hardware thread and mainly influences the energy-efficient turbo (EET) as well as the uncore frequency scaling (UFS) [18]. The EPB can be set via a machine-specific register (MSR) to *powersave*, *balanced*, or *performance* mode. To evaluate the decisions made by the EPB, we start with executing a compute-intensive workload on all cores of a processor and change the frequencies of all cores from 1.2 GHz to the maximum frequency (3.1 GHz w/ turbo) for different EPB settings and measure the number of instructions retired (completed instructions) as well as the power consumption using RAPL counters.



(a) Compute-intensive workload with EPB set to *powersave* or *balanced*.



(b) Compute-intensive workload with EPB set to *performance*.



(c) Memory-intensive workload with EPB set to *powersave* or *balanced*.

Figure 4.8: Power consumption and instructions retired over time, while switching all cores from 1.2 GHz to maximum core frequency (at 1000 ms) for different workload types and EPB settings.

Figure 4.8(a) shows the behavior over time for the *powersave* or *balanced* EPB setting. The frequency change happens at the 1000ms time point. We observe that the CPU immediately draws more power and sticks for about 1.2 s at 2.6 GHz and afterwards enters the turbo mode (3.1 GHz), which causes an additional power draw and a significant performance increase. Repeating the same experiment having the EPB set to *performance*, lets the CPU immediately enter the turbo mode after the frequency change, avoiding the 1.2 s delay as shown in Figure 4.8(b). Figure 4.8(c) visualizes the results for the experiment with an EPB setting of *powersave* or *balanced*, this time using a memory-intensive workload. Here, we observe the same delay of 1.2 s before activating the turbo mode. However, the experiment demonstrates that this decision was a bad one in terms of energy efficiency, since the CPU invested a lot of power, but was not able to increase the performance as the instructions retired measurement indicates. Thus, the only impact of the EPB on the core frequency we could observe, was that a 1.2 s delay was added before entering the EET.

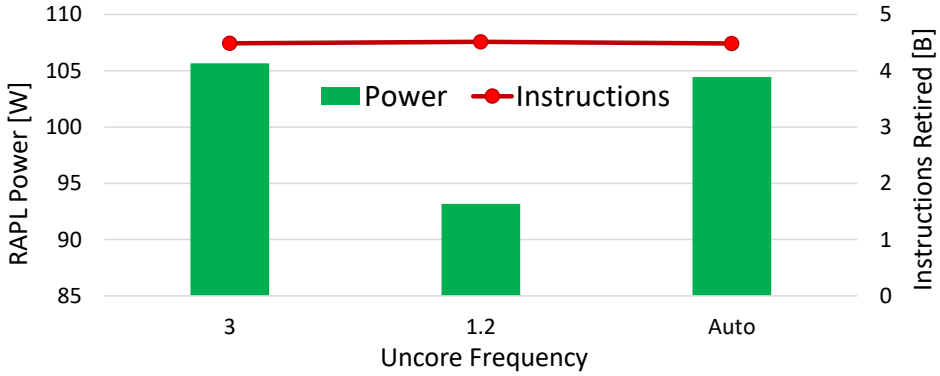


Figure 4.9: Throughput and power consumption of a compute-intensive workload for different uncore frequency settings.

To evaluate the impact of the EPB on the uncore frequency scaling (UFS) decisions, we run a compute-intensive workload with all cores running at maximum frequency and compare the measurements of power consumption and performance for automatic UFS as well as for having the uncore frequency pinned to 1.2 GHz and 3 GHz. Figure 4.9 shows the respective results. The number of retired instructions is the same for all uncore clock settings with a slight advantage for the lowest uncore frequency. Nevertheless, the automatic UFS decides to use the highest uncore frequency, which draws additional 12 W compared to the 1.2 GHz setting, which even delivers a bit more performance. Thus, this experiment once again confirms a bad decision making of the built-in CPU power management facilities and suggests to set the EPB to performance mode, when doing explicit energy control.

4.2.5 Summary and Conclusions

In this section, we analyzed performance and power characteristics of current mainstream server systems and focused on the available energy-control features and the energy-related decisions that are made by the CPU itself. Our analysis of the server system demonstrates that there are a lot of opportunities to save power on current hardware. These power savings can be achieved by appropriately configuring the offered energy-control knobs of the hardware, which significantly influence performance and power consumption of the system. Moreover, our evaluation also revealed that the decision making of the CPU in terms of power management is not sophisticated, actually, which once again motivates explicit energy-control in main memory database systems.

4.3 Energy Profiles

Based on our findings of the previous section, we will now abstract the different hardware energy-control facilities to configurations, which will be aggregated to an energy profile. The energy profile is a substantial component of the *ECL*, since it represents performance and energy efficiency trade-offs for the current workload. In this section, we discuss the configuration generation process and how the energy profile is related to the current workload of the DBMS.

4.3.1 Configuration Generation

A configuration represents a specific system state in terms of hardware energy-control settings for a single processor. Configurations are workload-agnostic showing different performance and energy characteristics when being evaluated in the context of a specific workload. A single configuration comprises:

The Set of Active Hardware Threads. This set defines which hardware threads of the processor are active. In the context of ERIS, this set defines which *Living Partition Vitalizers (LPV)* are running within this configuration. Since the processors of our target system have homogeneous cores, we allocate hardware threads from the left to the right. Nevertheless, physical cores usually support simultaneous multithreading (e.g., via HyperThreading) and thus, we use a HyperThread-first allocation strategy, which turned out to be the most energy-efficient one [109].

The Core Frequencies. This set defines the frequencies of the active physical cores of the configuration. Within the used hardware setting, all HyperThreads of a physical core share the same frequency. All other frequencies are set to their respective minimum.

The Uncore Frequency. If the hardware platform supports a separate frequency for the uncore part of the processor, like our test system, the respective uncore frequency for the configuration is specified here.

Due to our experiments regarding the EPB, the EPB is always set to *performance* for all configurations, because it only seems to delay the activation of the turbo mode. Hence, a configuration is expressed as:

$$c_x = (\{\text{core}_{\text{HyperThread}}\}, \{(\text{core}, f_{\text{core}})\}, f_{\text{uncore}}) \tag{4.1}$$

For instance a configuration c_1 can be instantiated as:

$$c_1 = (\{1_1, 1_2, 2_1\}, \{(1, 1.2\,\text{GHz}), (2, 2.1\,\text{GHz})\}, 3\,\text{GHz}) \tag{4.2}$$

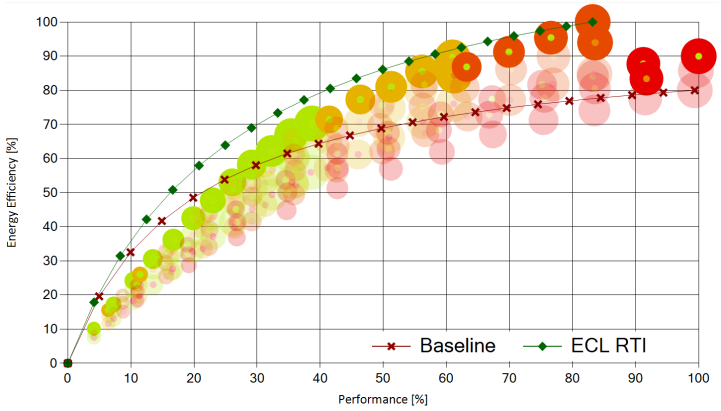
This configuration activates the first physical core and both of its HyperThread siblings as well as the second physical core with one HyperThread. The core frequency of the first physical core is set to 1.2 GHz and the second core clock is set to 2.1 GHz. The uncore clock of the processor is pinned to 3 GHz. To put a configuration into the context of a specific workload, it needs to be evaluated. During this evaluation process of a configuration, the configuration is enriched with the following information:

The Power Consumption. As we have shown in Section 4.2, the actual power draw of a configuration is workload dependent. Hence, this information specifies the power consumption of the configuration for a specific workload. The power consumption is measured for the entire processor by the RAPL counters including the Package and DRAM domain.

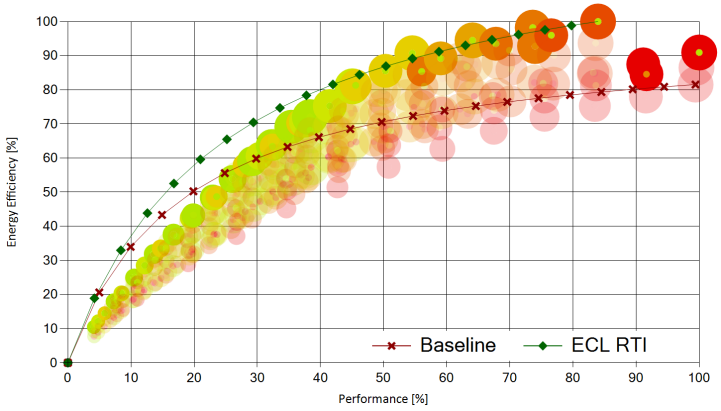
The Performance Score. The performance of a configuration depends on the particular workload the database system is executing and vice versa. As we have discussed in Section 2.3.1, the choice of the appropriate unit for performance measurements highly depends on its application. Hence, we compared transaction-level and instruction-level measurements for different workloads and observed a high correlation between them. Since instruction-level measurements are more fine-grained in terms of their update frequency, we decided to use the number of instructions retired by all of the active hardware threads on the processor. This performance can be measured with the help of the integrated performance counters of the processor.

The Energy Efficiency. To express the actual trade-off between performance and power consumption, we use the energy efficiency metric (cf., Section 2.3.2), which is calculated as performance score divided by power consumption.

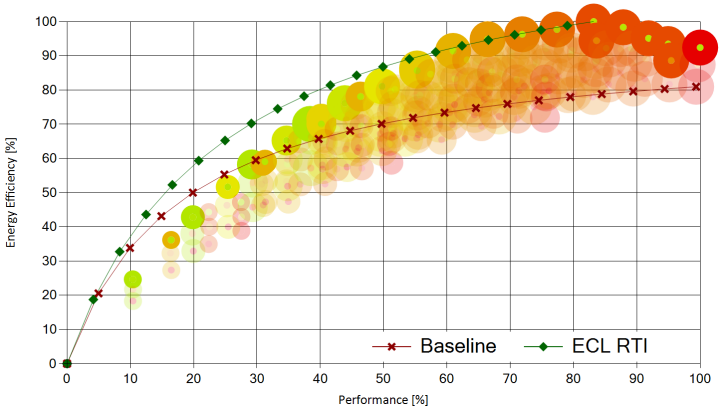
We call a set of configurations the *energy profile*. The configuration generator is responsible for systematically identifying a set of configurations that covers a high variety of distinct system states to explore most of the configuration spectrum. In Figure 4.10, we compare three different parameter settings of the configuration generator. The available parameters are the number of different core frequencies $|f_{core}|$, the number of distinct uncore frequencies $|f_{uncore}|$, the usage of mixed core frequencies $f_{core-mixed}$ (either enabled or disabled), and the maximum number of



(a) $|f_{core}| = 4$, $f_{core-mixed} =$ off, $|f_{uncore}| = 3$ (145 configurations).



(b) $|f_{core}| = 7$, $|f_{uncore}| = 3$, $f_{core-mixed} =$ off (252 configurations).



(c) $|f_{core}| = 4$, $|f_{uncore}| = 3$, $f_{core-mixed} =$ on (207 configurations).

Figure 4.10: Energy profiles using a compute-intensive workload for different configuration generator parameter settings. c_{max} set to 256.

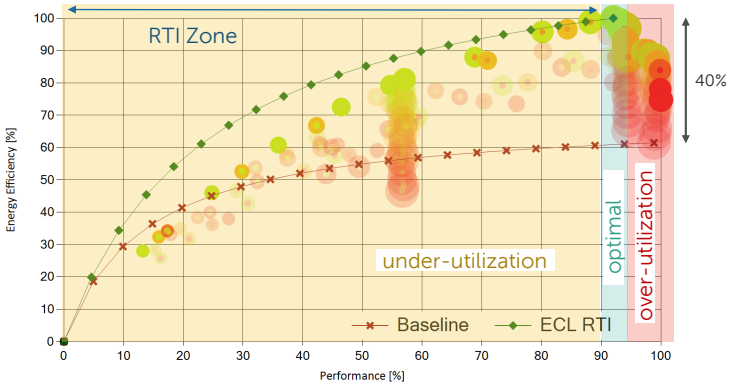
generated configurations c_{max}. Using these parameters, the configuration generator calculates all unique configurations taking the homogeneity of the individual cores into account, for instance activating physical core 1 is the same as activating core 2 regarding its performance and power characteristics. If the resulting number of configurations is too high, the generator aggregates hardware threads to groups resulting in a decreased granularity of the energy profile.

For our experiments we set c_{max} to 256 for visualization reasons. Figure 4.10(a) shows the energy profile for $|f_{core}| = 4$ (including the lowest, highest, and turbo frequency), $|f_{uncore}| = 3$, and $f_{core-mixed} =$ off while the DBMS runs a compute-intensive workload. The resulting number of configurations is $|cores| \cdot |f_{core}| \cdot |f_{uncore}| = 288$. Because c_{max} is limited to 256, the generator treats both HyperThread siblings of a physical core as one core group resulting in 144 configurations plus the idle configuration. The color of the outer circles encodes the average core frequency, the inner color the uncore frequency, and the diameter the number of active cores. The profile shows that the lowest frequencies are the most energy-efficient ones for low performance levels until their respective performance potential is exhausted. Moreover, we observe that the lowest uncore frequency is the most energy-efficient one, as it is supported by the experiment in Figure 4.9.

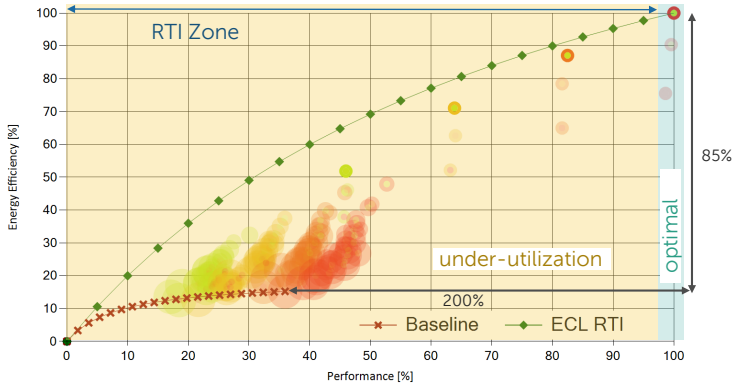
Using the energy profile, the ECL can determine the most energy-efficient configuration for a specific demanded performance level. Thus, only the skyline of the profile is of interest (opaque circles in the chart). Database systems without energy-control mechanisms usually use all available cores at the highest frequency as long as enough work is available, which is known as race-to-idle (RTI). Therefore, the baseline in the figure shows the respective energy efficiency that is achieved for different performance demands using this approach (cf., Section 2.3.3). Obviously, a more energy-efficient way is using an RTI strategy that switches between idle mode and the most energy-efficient configuration, which is depicted as the ECL RTI line. Increasing $|f_{core}|$ (Figure 4.10(b)) to 7 or enabling $f_{core-mixed}$ (Figure 4.10(c)) is not significantly improving this skyline, but causes the profile to include more configurations, which are more costly to maintain in case of a workload change. The skyline is not improving, because the original parameter setting already covered the most important supporting points of the configuration space.

4.3.2 Workload Dependency

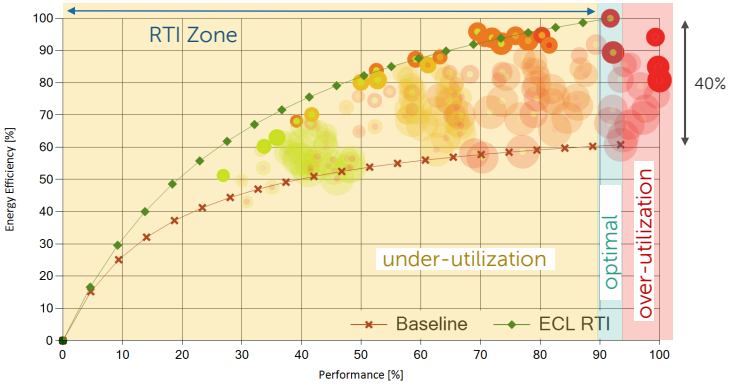
As mentioned before, the main purpose of the energy profile is to provide the ECL with information about performance level and power consumption of specific configurations. However, especially the given performance of a configuration heavily depends on the current workload of the database system. While the compute-intensive workload in Figure 4.10 shows an almost perfect profile, because there are no bottlenecks present in the system, real-world profiles look much more different. This difference occurs usually as soon as the contention on resources increases. For an in-memory DBMS, those points of contention are usually the memory controller or shared cache lines.



(a) Memory-intensive workload.



(b) Shared counter increment.



(c) Shared hash table insert.

Figure 4.11: Energy profiles using different workloads. $|f_{core}| = 4$, $f_{core-mixed}$ = off, $|f_{uncore}| = 3$ (145 configurations). The different ruling zones for the *Node ECL* are highlighted.

To demonstrate the effect of memory controller contention, we conducted the energy profile for a memory-intensive workload (i.e., a column scan), which is shown in Figure 4.11(a). This profile looks completely different compared to the one of the compute-intensive workload. It shows that high frequencies are a bad choice, because the bandwidth can not be further increased, and that a high uncore frequency is beneficial in terms of performance and energy efficiency (the dense clusters in the profile).

To quantify the impact of cache line contention causing bottlenecks, we used a workload where all threads atomically increment a single variable. As Figure 4.11(b) shows, the profile once again looks completely different. Here, the most performing and most energy-efficient configuration uses only two hardware threads at turbo frequency with the lowest uncore frequency. While the maximum possible energy savings amount up to 40 % (highest difference between baseline RTI and ECL RTI) for the previous workload, the savings in terms of energy are about 90 % in such a scenario with an additional query response advantage of 200 %. Nevertheless, since such a workload is very artificial and uncommon for real-world workloads, we show an additional energy profile in Figure 4.11(c) that was conducted using a workload where multiple threads insert values into a shared hash table. We again observe the same effects at a smaller scale with a potential energy saving of 40 % and a query response benefit of about 8 % compared to the baseline.

Based on the experiments, we can conclude that the shape of the energy profile can change arbitrarily for different workloads, because contention on hardware resource is the common case in main memory database systems. Moreover, we have shown that choosing the right configuration can significantly improve the energy efficiency as well as the response time.

4.3.3 Energy Profiles and the ECL

The node-specific energy profile is the most essential component of the ECL, because it contains important information about the available hardware configurations as well as their performance and energy metrics. As depicted in Figure 4.11, the ECL differentiates three different high-level zones for energy ruling:

The Optimal Zone. This zone includes only the most energy-efficient configuration and the ECL is eager to reside in this zone, because the most energy savings are experienced here.

The Under-Utilization Zone. Since database severs are mostly over-provisioned to cope with load peaks, most of the time is usually spent in this zone where energy efficiency of the configurations is mostly significantly lower compared to the optimal zone. Thus, the ECL will use the race-to-idle method within this zone, which means that the ECL is frequently switching between idle mode and the most energy-efficient configuration (optimal zone). The potential energy savings for this method are shown by the ECL RTI line. Using ECL RTI, the ECL is able to partially compensate the high energy costs for activating the

first core on a socket (cf., Figure 4.5). The ECL RTI energy savings are about 40 % (RAPL only) for very low performance demands and decrease the more performance is needed, because the time the system is able to reside in idle mode is decreasing.

The Over-Utilization Zone. The configurations of this zone are only applied, if the optimal zone does not provide enough performance to master the current load within the given query response time constraints.

The respective range of the individual zones depends on the energy profile and thus, on the workload. For instance, for the energy profiles in Figure 4.11(b) and 4.11(c), the *over-utilization zone* is very small or not even present.

4.3.4 Summary and Conclusions

In this section, we specified the concept of *Configurations*, which represent a specific system state in terms of hardware energy-control settings for a single processor and thus, the set of active LPVs ERIS is running. Configurations are evaluated in the context of a specific workload to be enriched by information about the power consumption, the delivered performance, and the effective energy efficiency.

A set of configurations is aggregated to an *Energy Profile*. This set of configurations is generated with the help of a configuration generator, which tries to cover the most important supporting points of the big exploration space. As we have shown, the cardinality of the configuration set can be kept low, while still reaching a good quality of the energy profile. Moreover, we demonstrated that the shape of the energy profile is highly workload dependent and that this shape affects the decision making of the ECL.

4.4 Resource Adaptivity-Specific Energy-Control Loop

In this section, we present the details of the *Resource Adaptivity*-specific Energy-Control Loop (ECL). This implementation of the ECL is designed hierarchically as depicted in Figure 4.1. Each processor independently runs a *Node ECL* maintaining its own *Energy Profile* and is only configuring the hardware resources available on the respective CPU. Besides the Node ECLs, a *Global ECL* is responsible for monitoring the query response times and influences the decision making of the individual Node ECLs. Compared to our big picture of the ECL in Figure 2.18, *Resource Adaptivity* implements the CPU-level (Node ECL) as well as the System-level (Global ECL) of the overall hierarchy. In the following, we discuss both components in detail including their integration into ERIS.

4.4.1 Node ECL

Since each CPU possesses its own energy counters (RAPL), the lowest available unit for measurements is a single processor (Node). Thus, it is a natural decision to rule

the energy tuning features at the CPU-level, which is exactly the job of a Node ECL. For that reason, there are as many active Node ECLs as processors available on the platform, each of them running as a separate thread, that is pinned to the respective processor. Because the workload characteristics can vary per processor, each Node ECL uses and maintains its own energy profile to achieve a high accuracy.

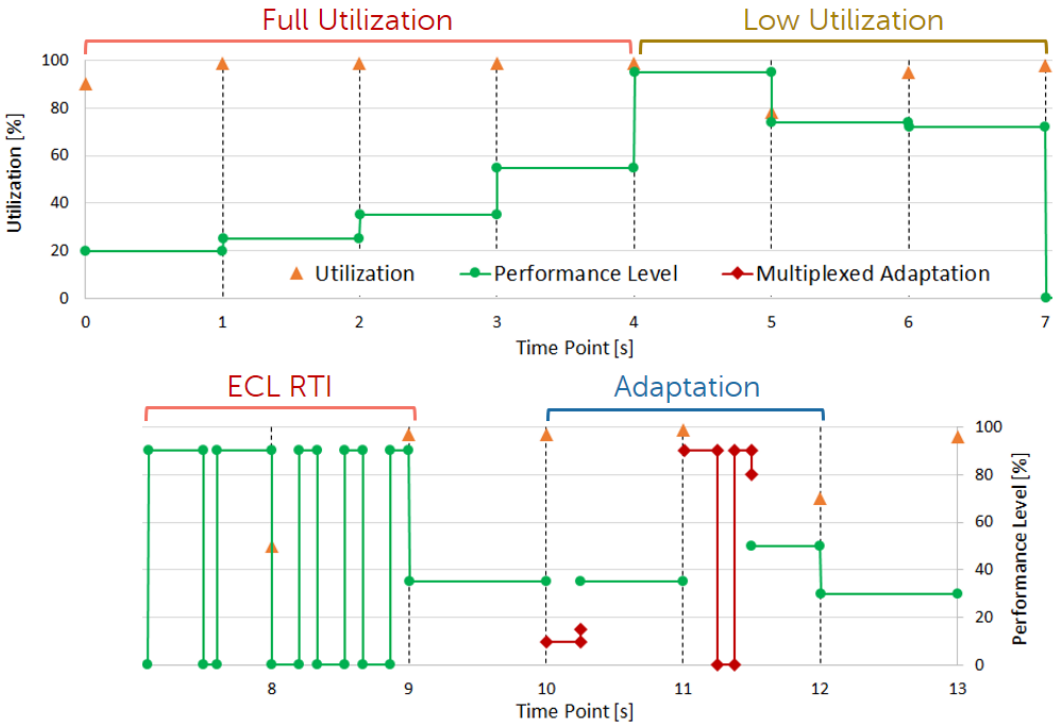


Figure 4.12: Guiding Node ECL control mechanism example.

Decision Making

The entire ECL as well as the Node ECLs, are designed as a reactive control loop. Thus, the Node ECL needs to quickly respond to load changes and is therefore executed periodically at the scale of a second or less. Figure 4.12 shows a guiding example that we will leverage throughout the discussion of the Node ECL. The example shows the measured *utilization* of the runtime and the *performance level* that is applied by the Node ECL over time. In ERIS, the utilization of a processor is measured by monitoring the amount of time the running LPVs actually do work, i.e., in the role of a task, as an LP, or as node coordinator. The base Node ECL interval is set to one second in the example. As visualized in Figure 4.13, the Node ECL consists of two main components, which we will discuss in the following:

The Utilization Controller. This controller is responsible for determining the current performance demand of ERIS on the specific processor. Hence, the

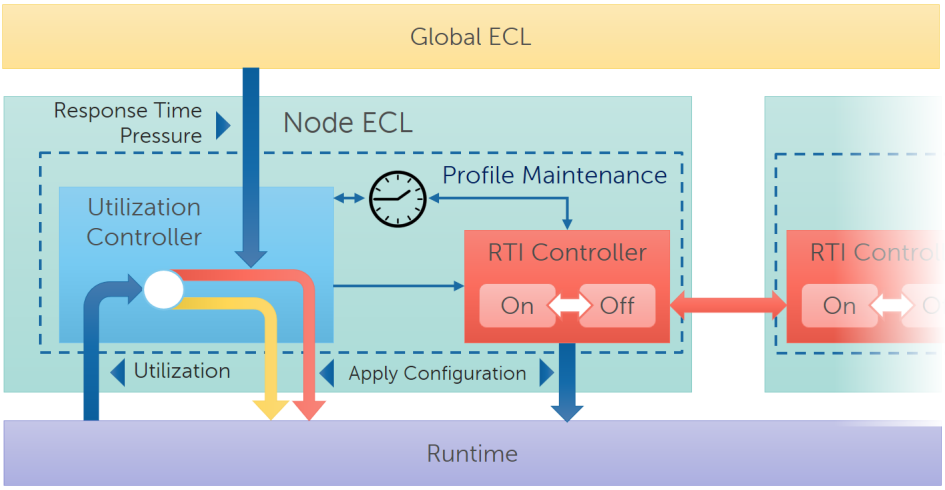


Figure 4.13: Control mechanism of the Node ECL.

utilization controller continuously receives utilization information from the database runtime. As shown by the example in Figure 4.12 at the time points 1 to 4, the database runtime reports a full utilization (100 %) forcing the utilization controller to increase the performance level and apply the corresponding most energy-efficient configuration fulfilling the calculated performance demand based on the energy profile. Since the utilization can only be measured relative to the amount of active hardware threads, the utilization controller is not able to exactly determine the required performance level in case of a full utilization. Thus, the utilization controller uses a discovery process that exponentially increases the performance level in each Node ECL call to avoid activating more hardware resources than necessary on the one hand and to cope with load spikes on the other hand. Additionally, the discovery process considers information reported from the Global ECL. The opposite scenario is a utilization below 100% as it happens for instance at the time points 5 and 6 in the guiding example. In this scenario, the utilization controller is able to quickly determine the required performance level using this formula:

$$\text{performance}_{\text{new}} = \text{utilization} \cdot \text{performance}_{\text{old}} \tag{4.3}$$

Once again, the Node ECL uses the calculated performance level and applies the most energy-efficient configuration satisfying the performance level, which is known by the energy profile.

The RTI Controller. The RTI controller leverages the information reported by the utilization controller (i.e., new performance level and configuration) and decides whether to use a race-to-idle strategy or not. There are two reasons for applying RTI. First, to partially compensate the high costs for activating

the first core on a socket and second, to emulate any performance level for which no configuration is known by the energy profile. In practice, the RTI controller always tries to switch between the most energy-efficient configuration and idle mode when the performance demand is in the under-utilization zone. However, the negative side effect of applying RTI is that the query response time is negatively affected, if the system resides for a long time in idle mode. Thus, the RTI controller switches at a high frequency (e.g., every 10ms) or selects a lower performing configuration for doing RTI. Additionally, the RTI controllers of different nodes try to synchronize idle times, because a socket can only enter its deepest sleep state, if all sockets of the machine are in idle mode (cf., Figure 4.5). In the example, we schematically demonstrate the RTI usage between time points 7 and 9. At time point 7, the configuration of the previous interval is emulated using RTI switching between the most energy-efficient configuration available (optimal zone) and the idle mode. At time point 8, the utilization controller detects a lower utilization and reduces the performance level accordingly. Hence, the RTI controller causes the system to spend more time in idle mode and uses three instead of two RTI cycles per Node ECL interval to keep the response time low. In practice, the RTI controller does up to 50 RTI cycles per 1s Node ECL interval.

Energy Profile Maintenance

As previously demonstrated, the shape and skyline of the energy profile depends on the current workload the DBMS is facing. Thus, the ECL needs to quickly adapt the energy profile in case of a changing workload. If the energy profile is not accurate, the Node ECL is not able to calculate the current performance demand, RTI calculations become inaccurate, and possibly energy-inefficient configurations are applied. For that reason, it is important how fast configurations can be reevaluated at runtime. Since the speed of applying new configurations and measuring the corresponding energy and performance counters is hardware dependent, the ECL does a meta calibration step on startup.

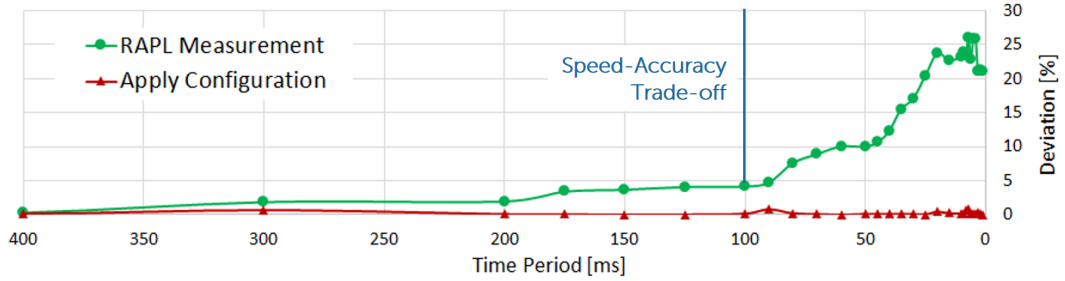


Figure 4.14: Accuracy of configuration evaluation for different time intervals.

In this meta calibration step, the ECL detects the times needed for *applying* a configuration and for *measuring* the counters. The ECL starts by taking a reference

measurement using a generous amount of time and is decreasing the times step-by-step, while measuring the deviation from the reference measurement. This process happens first for the *measure* time followed by the *apply* time. During the measurement, the ECL switches between the highest configuration (all cores at highest frequency) and the lowest available configuration (one core at lowest frequency). As presented in Figure 4.14, for *applying* a configuration, the evaluation is even accurate when using a 1ms interval, but the time for *measuring* the counters becomes more and more inaccurate when being decreased. The source of most of the deviation we encountered, was the RAPL measurement, when switching to the lowest configuration. We identified a *measurement* interval of 100 ms to be the best trade off between accuracy and speed for our hardware environment. To adapt the energy profile at runtime, we use two strategies:

The Online Adaptation. This energy profile adaptation strategy is continuously used to adapt to slight workload changes. Every time the Node ECL applies a certain configuration, the Node ECL measures the power and performance metrics using the respective performance counters (i.e., RAPL and instructions retired) and updates the energy profile. The main advantage of this strategy is that almost no overhead is generated and the currently used configurations are highly accurate. However, the obvious drawback is that only configurations are maintained, which are reported by the energy profile to be the most energy-efficient ones, which is may not the case anymore.

The Multiplexed Adaptation. This energy profile adaptation strategy overcomes this disadvantage. The strategy is triggered by the *Online Adaptation* as soon as a high drift in configuration accuracy is detected and reevaluates all configurations of the energy profile. To only minimally affect the operation of the Node ECL and to obey the query response time constraint, this strategy uses time-division multiplexing. Within one Node ECL period (e.g., 1 s), about 100 ms are used to evaluate a random configuration and in the remaining 900 ms, the Node ECL operates as usual (between time point 10 and 11 in Figure 4.12). During the usual operation phase, the Node ECL is able to respond to the intrusion of the adaptation process using its energy profile that is getting more accurate over time. To evaluate high performing configurations, the multiplexed adaptation once again leverages the capabilities of the RTI controller to simulate high load situations, which can enlarge the time fraction that is used for adaptation within one Node ECL period (between time point 11 and 12 in the example). It is also possible that a configuration is reevaluated using an evaluation time shorter than 100 ms (i.e., if the system load is very low). In such a situation, the Node ECL tolerates the less accurate energy and performance metrics, which will be corrected by the *Online Adaptation* process, as soon as the respective configuration is applied.

As discussed, both energy profile adaptation strategies work hand in hand and expose a different behavior in terms of overhead, invasiveness, and quality of the

resulting energy profile. As we will show in our end-to-end evaluation, even the *Multiplexed Adaptation* only minimally affects the operation of the ECL and the combination of both strategies successfully maintains the energy profile in the background.

4.4.2 Global ECL

While Node ECLs are responsible for managing a single socket of the system, the Global ECL manages metrics that are only globally available. In our scenario, this metric is the average query response time, which is the result of the performance of all available sockets in the system. Thus, all Node ECLs implicitly influence this metric. In ERIS, the average query response time is measured by timing the execution speed of a task. The overall goal of the Global ECL is *not* to keep the response time as low as possible. Instead, it takes a user-defined maximum average response time as an instance of a QoS measure, which is considered a soft constraint, because a reactive control loop is not able to guarantee that this limit is not violated and thus, uses a best effort strategy.

Hence, the Global ECL continuously monitors the actual average query response time and calculates its current trend. Based on the trend, the Global ECL is able to estimate the time until the response time limit is violated. This time is provided for all Node ECLs, which use this value to adjust the aggressiveness to enter a higher performing configuration in case of a full utilization and to adjust the RTI usage, since RTI negatively affects the response time. However, a low value of this time (or even zero if the limit is already violated), does not mean that the *Node ECL* automatically ramps up all available hardware resources, because the work can be unevenly distributed across the sockets. Thus, Node ECLs still select lower performing configurations, if a lower utilization is reported, but are more eager to increase the performance level on the respective processor.

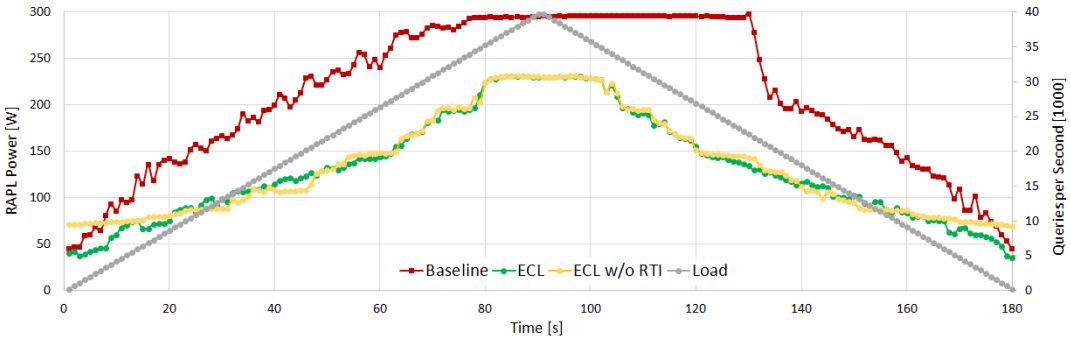
4.5 End-to-End Evaluation

In this section, we present an end-to-end configuration of ERIS including an implementation of *Resource Adaptivity*. According to our definition for benchmarking the energy awareness of a DBMS (cf., Section 2.4.2), we evaluate the resource adaptivity-specific ECL for different workloads as well as load profiles and we present experiments regarding the ability of the ECL to adapt the energy profile to a changing workload. All experiments are conducted on the 2-socket Xeon E5-2690 v3 presented in Section 4.2.1 running the in-memory DBMS.

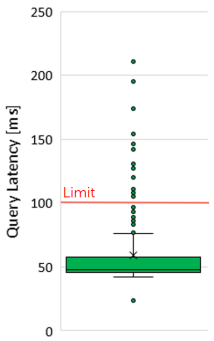
4.5.1 Workload and Load Profile

To investigate the behavior of the ECL and the respective energy savings, we ran experiments for the combinations of two different workloads and two load profiles. The workload specifies the types of queries that are sent to the DBMS and the

load profile defines the number of queries per second over time sent to the database system. We use two different workloads querying a key-value store that is loaded with 20 M keys (SF 200), which is enough to fully utilize the memory controllers when scanning over the keys only. While the first workload *scans* over the key column to find the requested key and is thus, memory-bound, the second workload uses an *index* to find the corresponding key and is therefore, intentionally *not* memory-bound. Both workloads simulate typical data object access patterns and have a completely different energy profile, for instance, the energy profile of the scan workload resembles the one in Figure 4.11(a). As load profile we use the *spike* profile, which is easy to understand and is thus suitable to explain the ECL behavior for different load situations. Additionally, we use a *twitter* [1] load profile, to investigate how the ECL acts in real-world system load situations.



(a) Load profile and power consumption of the baseline and the ECL (2 sockets).



(b) Query latency.

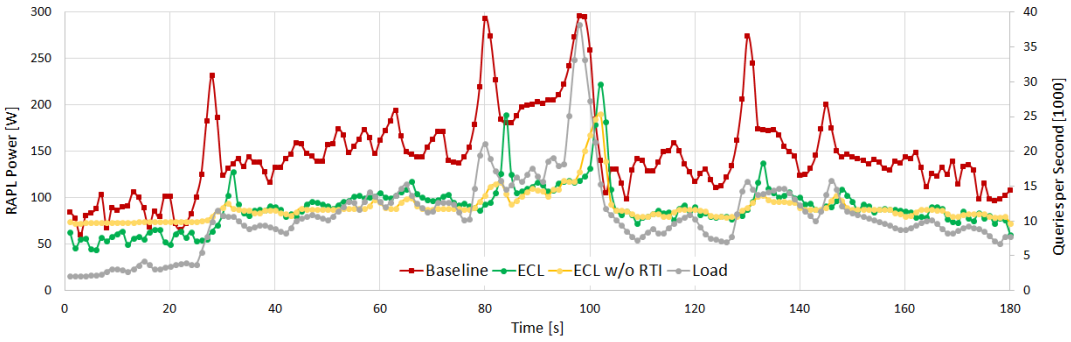
Figure 4.15: Power consumption and query latency over time for the spike load profile (column scans).

Figure 4.15 shows the results for the combination of the *scan* workload and the *spike* load profile, which was run for 3 minutes. The respective system load over time is visualized in Figure 4.15(a). Additionally, the figure includes the RAPL power measurements over time for the baseline (race-to-idle using all available hardware resources with CPU and OS frequency control), the ECL with RTI enabled, and the

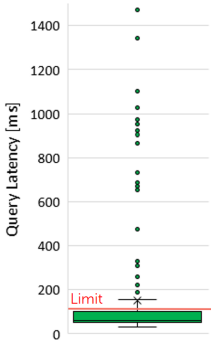
ECL with RTI disabled. The first observation we made is that the ECL never draws more power than the baseline, except for very low load situations when having RTI disabled, because the ECL never enters idle mode in this setting. The next finding is that the ECL significantly improves the energy proportionality, especially in load situation above 50% where we see almost perfect energy proportionality, except for the overload situation between 80 and 100 s. In lower load situations (<50 %), energy proportionality gets worse because of the high static power consumption of the processors, which gets less significant in higher power levels. Starting at 80 s, the load profile generates an overload situation meaning that the system receives more queries than it is able to actually handle. An interesting observation is that the baseline stays for about 50s in the overload state, while the ECL only resides for about 20 s in the overload situation and still draws less power. The reason for this effect is, that using all available hardware resources (baseline) provides less performance compared to the performance of the configuration that is selected by the ECL, because more contention on hardware resources is generated. As shown by Figure 4.15(b), the ECL was able to stay within the response time limit of 100 ms most of the time and the response time limit violations happened within the overload situation.

In Figure 4.16, we present our results for the combination of the *scan* workload and the *twitter* load profile. Compared to the *spike* load profile, this profile includes sudden load peaks and is frequently changing between increasing and decreasing the system load. The behavior of load and power consumption for the baseline and the ECL (with and without RTI) over time are shown in Figure 4.16(a). Once again, we observe that the ECL is drawing significantly less power compared to the baseline most of the time. However, we also observe that the ECL takes more time to adapt the hardware configuration to the sudden load peaks, because of its reactive nature. This finding is supported by the average response time measurements depicted in Figure 4.16(b). Here, we see that the ECL is able to stay within the response time limit most of the time, but we also observe outliers, which occur during load peaks.

Our overall goal is to reduce the overall energy consumption of the database system. Hence, we measured the energy consumption of all workload and load profile combinations for the baseline and for the ECL and calculated the energy savings of the ECL. The results are shown in Figure 4.17. The numbers show that the ECL is able to save up to 38.6 % energy compared to the baseline when using the *scan* workload. For the *index* workload the ECL is able to save up to 22.7 % energy. Thus, the energy saving potential mainly depends on the workload type as we already observed in Section 4.3.2 (energy profiles). Since RAPL and PSU power measurements are highly accurate on this platform [50], those numbers, which were measured using the RAPL counters, also apply to the whole server. Moreover, we observed that the ECL itself only consumes 2 % of the compute time of a single hardware thread per socket, which is a negligible number.



(a) Load profile and power consumption of the baseline and the ECL (2 sockets).



(b) Query latency.

Figure 4.16: Power consumption and query latency over time for the twitter load profile (column scans).

4.5.2 Energy Profile Maintenance

While the previous experiments used a static workload type, we will now evaluate how the ECL responds to workload changes that occur at runtime. To do so, we set the load to a static value (50 % of the peak performance) and suddenly switch from the *index* workload to the *scan* workload. We use three ECL settings to demonstrate the effect of the different energy profile maintenance strategies. The first setting performs no energy profile maintenance (*ECL static*). The second setting uses only the *Online Adaptation* (*ECL online*) and the last one additionally uses the *Multiplexed Adaptation* (*ECL multiplex*).

Figure 4.18 visualizes the power consumption over time for the different adaptation strategies. The workload switch happens at 40 s. When looking at the *ECL static* measurements, we already observe a slightly increased power consumption before the workload switch, because online adaptation is disabled and the profile is not adjusted to small variations that usually occur even for a static workload and mostly originate from the hardware itself. Immediately after the workload switch, we see that the power consumption is higher compared to the other maintenance strategies and

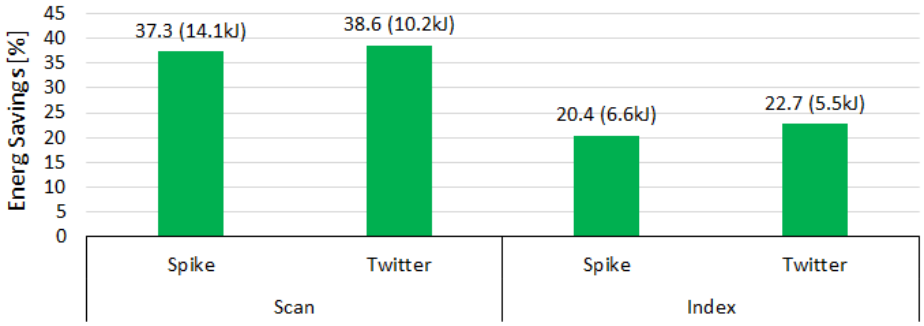


Figure 4.17: Energy savings for different load profiles and workload types.

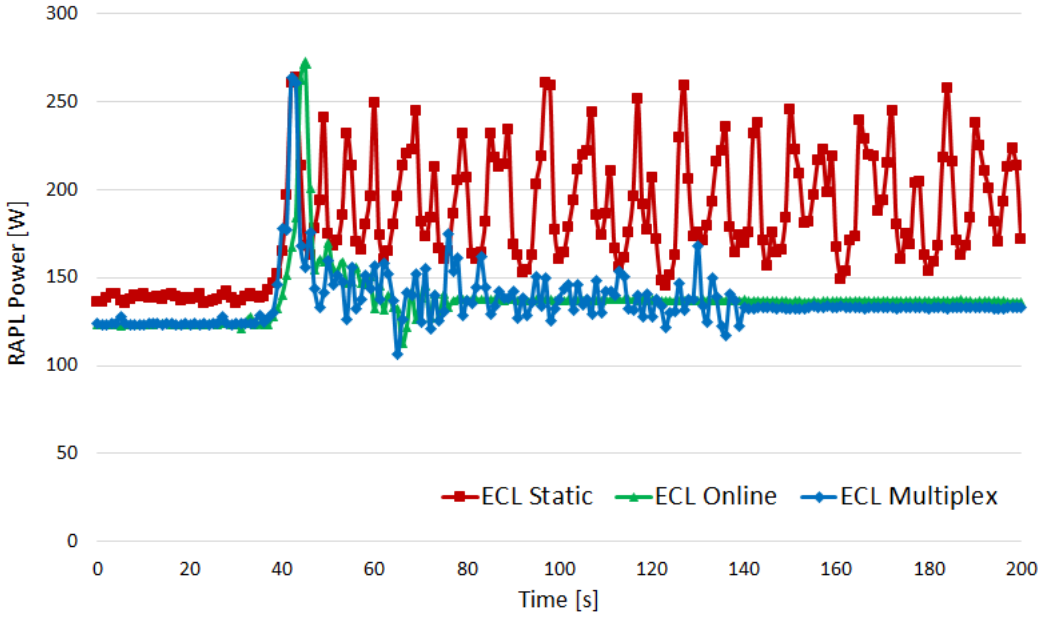


Figure 4.18: Power consumption during workload switch for different energy profile maintenance strategies.

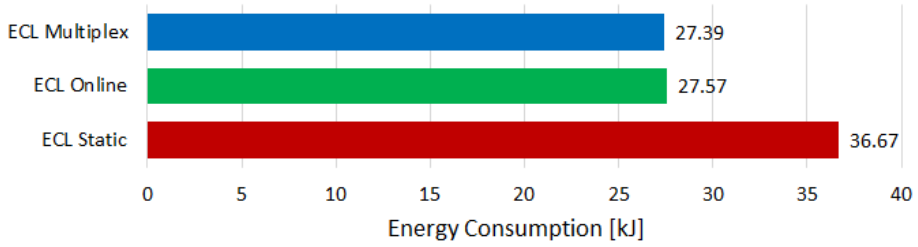


Figure 4.19: Total energy consumption for workload switch.

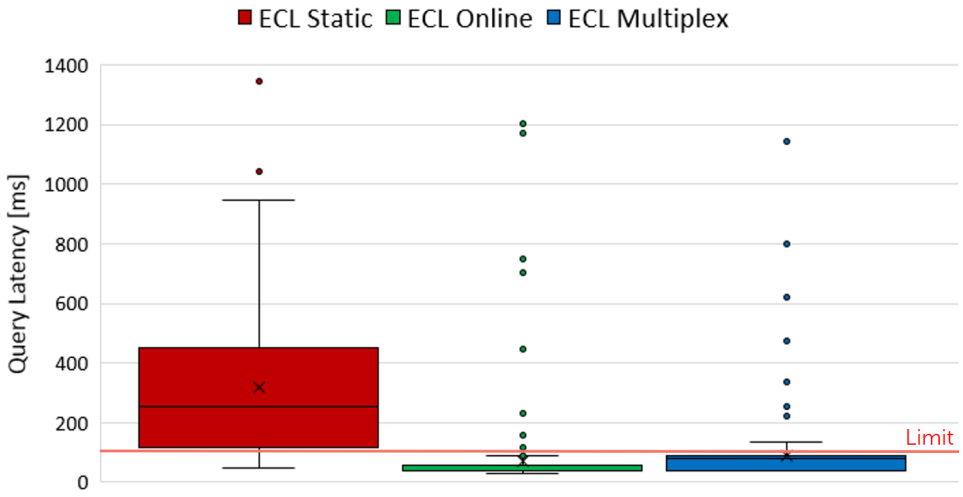


Figure 4.20: Query latency for workload switch.

that the numbers are fluctuating, because the Node ECLs are not able to accurately determine the power demand, to find the appropriate configurations, and to do accurate RTI calculations. The *ECL online* measurements show that the online adaptation quickly adapts the energy profile allowing the Node ECLs to do accurate calculations and configuration decisions. However, since the online adaptation only reevaluates configurations that were applied, it is necessary to reevaluate all the other configurations of the energy profile, which is done by the multiplexed adaptation. As the *ECL multiplex* measurements show, this process requires more time, but manages to find a slightly more energy-efficient configuration for the new workload.

Figure 4.19 shows the corresponding total energy consumption measurements for the three energy profile maintenance settings. As assumed, the *ECL static* setting without any energy profile adaptation draws significantly more energy and is mostly not able to stay within the query response time limit as shown in Figure 4.20. By contrast, the *ECL online* and *ECL multiplex* settings consume about 25 % less power and are able to stay within the response time limit. Thus, we can conclude that the active energy profile maintenance at runtime is very important for a static workload as well as for changing workloads.

4.6 Summary and Conclusions

In this chapter, we presented *Resource Adaptivity* as the hardware-centric implementation of our *Energy Awareness by Adaptivity* concept. While previous research mainly focused on disk-based DBMSs, resource adaptivity aims at investigating and optimizing the energy consumption of highly parallel state-of-the-art in-memory database systems that make heavy use of the main power consumers – CPUs and main memory – and are thus, an attractive target for energy optimizations. Our in-depth energy analysis of a current server system showed that modern processors provide a rich set of energy-control facilities, but lack the capability of controlling them appropriately.

Afterwards, we specified the concept of *Configurations*, which represent a specific system state in terms of hardware energy-control settings for a single processor and thus, the set of active LPVs ERIS is running. Configurations are evaluated in the context of a specific workload to be enriched by information about the power consumption, the delivered performance, and the effective energy efficiency. A set of configurations is aggregated to an *Energy Profile*. This set of configurations is generated with the help of a configuration generator, which tries to cover the most important supporting points of the big exploration space. As we have shown, the cardinality of the configuration set can be kept low, while still reaching a good quality of the energy profile. Moreover, we demonstrated that the shape of the energy profile is highly workload dependent.

Finally, we proposed the resource adaptivity-specific ECL as a holistic software-based approach for adaptive energy-control on scale-up in-memory database systems that obeys a query latency limit as a soft constraint and actively optimizes energy efficiency and performance of the DBMS. Resource adaptivity effectively implements the CPU-level and the System-level of the overall energy-control loop by employing a Node ECL per processor and a Global ECL. Node ECLs rely on adaptive workload-dependent energy profiles that are continuously maintained at runtime using the *Online Adaptation* and *Multiplexed Adaptation* maintenance strategies. In our evaluation, we observed energy savings of up to about 40 % for real world load profiles (cf., Table 4.17). Moreover, we demonstrated that the Node ECLs are able to quickly and efficiently adapt their energy profile in case of workload changes without inducing a significant overhead.

5 Storage Adaptivity

Modern application scenarios require data management systems to deal with a high variety of datasets and query types. Due to the rapid and agile development as well as frequently shifting focuses of interest today, neither the type and the schema of datasets nor the queries executed on them are known beforehand and additionally, both – datasets and queries – are constantly changing over time. Even in such a dynamic environment, the database system still needs to guarantee data velocity and responsive query processing in real-time to satisfy today's business needs.

Because the overall performance of a DBMS mainly depends on the physical data layout, modern database systems need to adapt their physical storage layout frequently without significantly hurting the query response time during the adaptation process to appropriately face the challenges of modern application scenarios. This adaptation process needs to consider data characteristics as well as query patterns and hot spots that are subject of continuous change. The huge design space for a specific instance of a physical storage layout includes the index selection, the choice of a representation (e.g., columnar, row-wise, or interpreted) per attribute respectively attribute group, and the redundant storage of attributes in different representations. Moreover, the specific physical storage layout can be inhomogeneous across the partitions of a single relation. We do not consider materialized views, because they are logical access paths that are out of scope of the low-level storage engine. Besides being able to store data in this highly dynamic way, the query processing engine needs to be able to deal with all of those mixed physical representations. For instance, mostly all current database systems store and process data either row-wise (e.g., MySQL) or columnar (e.g., MonetDB) and some DBMSs like HyPer [73] or IBM DB2 [61] only allow the user to organize a table either row-wise or columnar, but do not support mixes of both in a single table.

So far, research mostly focused on offline physical storage layout optimizations using design advisors [42, 139], which lack our agility and invasiveness – in terms of query response time – demands, because the full adaptation process is mostly manually triggered in times of a low system load. Recent approaches such as database cracking [63, 74] or adaptive storage engines for hybrid transaction-analytical processing (HTAP) workloads [11, 13] address the issue of either query-driven online index creation or switching between a row-wise or columnar physical storage layout respectively mixes of both representation. However, those approaches are only partial solutions for the entire online storage adaptation problem as shown in Table 5.1, which compares the features of the individual approaches.

In this chapter, we present *Storage Adaptivity* as a software-centric approach for increasing the energy awareness of a database system Our storage adaptivity is a

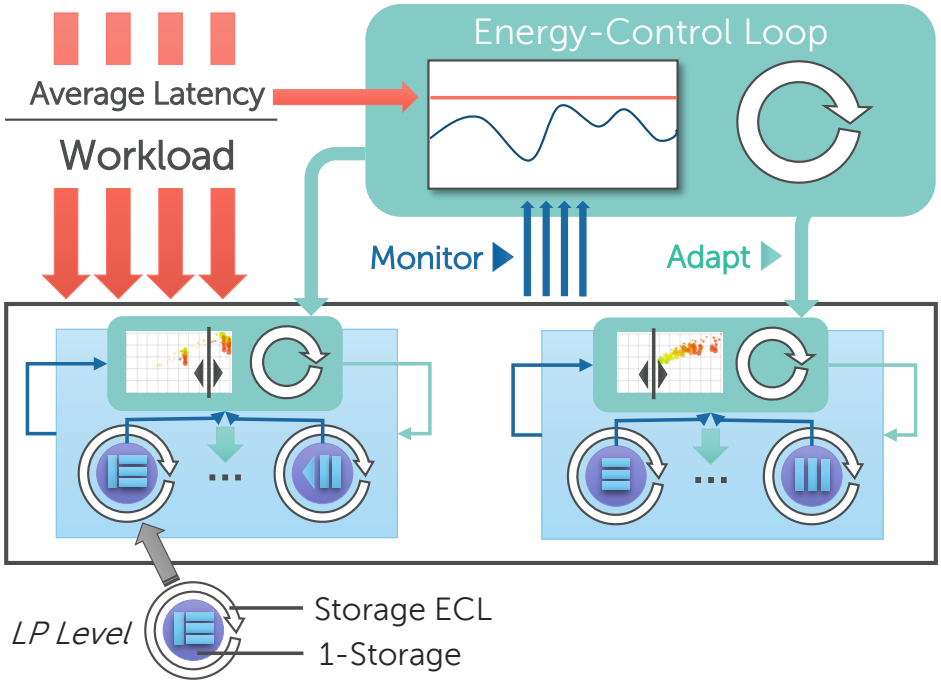


Figure 5.1: ECL hierarchy including the storage ECL per LP.

holistic approach for fine-grained physical storage layout adaptation at runtime that relies on two components as depicted in Figure 5.1. The first component is *1-Storage*, which is a storage manager that is able to store and process data in a high variety of physical storage layouts and is able to incrementally adapt the storage layout at runtime. 1-Storage employs a high-level concept to plug indexes, columnar, and row-wise worlds together and additionally considers schema flexibility. The set of available low-level storage modules (e.g., a specific column store or index implementation) is extensible and storage modules follow a fixed interface. Since 1-Storage is designed to operate within the *Living Partitions* architecture, each living partition uses its own 1-Storage and each of them is able to store its data in a different physical storage layout. The second component is the *Storage Adaptivity-Specific ECL*, which instantiates a *Storage ECL* per living partition and controls the adaptation process of the corresponding 1-Storage. As visualized in the overview Figure 5.1, this ECL type is at the lowest level of our ECL hierarchy and communicates with the higher level ECL types, i.e., the CPU-level Node ECLs and the system-level Global ECL, to consider the current hardware configuration (*Resource Adaptivity*) as well as current query latencies.

In the following, we will discuss the relevant related works followed by the presentation of the 1-Storage interface, which decouples operators from the actual physical storage layout. Afterwards, we will introduce the internal design of 1-Storage including the interface for storage modules and we will discuss the concept of micro

query execution plans in this context. In the next step we will present the storage adaptivity-specific ECL and describe how the specific physical storage layout for a 1-Storage instance is determined and how the actual adaptation is triggered and executed. Finally, we will exhaustively evaluate the different aspects of our storage adaptivity approach, especially in the context of energy awareness, and conclude this chapter.

5.1 Related Work

We split our discussion of the related work in three major parts. In the first part, we discuss the wide range of physical storage models that are known by research and are implemented in database systems. Here, we focus on the basic storage models as well as hybrid storage models – such as our 1-Storage – that are the foundation for any adaptive storage, because they define the degree of freedom that is is given for the adaptation. In the second part, we go through the offline physical storage layout tuning approaches, which are the predecessors of their online pendants. Finally, we discuss adaptive storage solutions that are able to adapt their physical storage layout at runtime. We will compare the capabilities of these approaches to our storage adaptivity approach, because they aim at the same goal.

5.1.1 Physical Storage Models

A physical storage model defines how data is organized in the byte-addressable main memory (volatile or non-volatile) respectively on block-oriented persistent drives (e.g., HDD or SSD). We distinguish between two basic storage models. The traditional basic storage model is the row-wise data organization also known as n-ary storage model (NSM) [36], which equals to the natural order where records are stored one after another. The counterpart to the row-wise organization is the columnar data organization also known as decomposition storage model (DSM) [36], which splits records into their individual attributes and the values of the same attribute are stored sequentially. While the NSM is a good choice for OLTP workloads, which access a high amount of attributes per query, the DSM has performance advantages for OLAP workloads that scan over a small amount of attributes, because the memory respectively disk bandwidth and caches are better utilized [23, 58]. Moreover, the columnar data organization exhibits better compression characteristics [7], but faces additional costs for tuple reconstruction. To reconstruct records in a columnar organized storage, all columns are either ordered by the sequential object identifier (OID) or each column value is associated with its OID. While the first version (positional addressing scheme) allows a fast tuple reconstruction, the second version (BATs) allows each column to be ordered differently to, for instance, speed up point queries using a binary search. Another way of speeding up point or range queries are auxiliary data structures like indexes, which have been exhaustively researched [79, 88, 116]. A modification of the NSM respectively the DSM for sparse data as it occurs in database systems that support schema flexibility are interpreted

records [17] or columns [8]. This modification adds memory and interpretation overhead, which amortizes at a certain sparseness ratio.

The first approach for a hybrid storage model is PAX [10], which organizes a row-wise storage like a columnar storage at the page-level. Here, physical memory pages are split into mini pages and each mini page sequentially stores the values of a specific attribute. The approach optimizes the cache usage, because values of the same attribute are not scattered across the entire memory page. Another hybrid approach are column groups respectively data morphing [55], which store attribute sets sequentially in memory that are often accessed in combination. Hence, column groups combine the advantages of the row-wise and the columnar storage model. For instance, the DBMS HYRISE [48] uses a column group-based storage model. Our 1-Storage approach also employs a hybrid storage model that allows row-wise and columnar organized data including their interpreted versions to coexist. Here, each attribute or sets of attributes are stored in one or more formats. Moreover, due to the implicit partitioning of the living partitions architecture 1-Storage is embedded into, each partition is able to use a different storage format and auxiliary data structures (e.g., indexes) mix that can be adapted at runtime.

Another issue is the tight coupling of physical storage model and query processing model to maximize the performance of operators. For instance, the HyPer DBMS [73] allows to either use a row-wise or a columnar storage model per relation, which also requires the implementation of two query processing models respectively operator code templates. An alternative approach is taken by fractured mirrors [114] that store tables redundantly in both basic storage models, which once again requires a row-oriented query processing engine for OLTP queries and a column-oriented engine for OLAP workloads (e.g., employed by Oracle). Our 1-Storage circumvents this issue by decoupling operators from the physical storage layout, which adds costs for the indirection layer, but allows full flexibility in the choice and mixture of storage formats. Nevertheless, we will investigate multiple ways of designing this indirection layer and point out how to get almost rid of the additional indirection costs.

5.1.2 Design Advisors

Design advisors either use a priori knowledge of the workload or use recordings of the recent workload to find the best performing physical storage layout. To actually apply the storage format modifications, the changes are either made in times of a low system load or are executed as background tasks, which is problematic in the presence of concurrent updates. In contrast, our *Storage Adaptivity* approach is able to incrementally adapt the physical storage layout on a fine-grained level at runtime even in situations of a peak system load where physical storage format adaptations are the only way to handle such critical situations without adding more hardware resources. Since auxiliary data structures like indexes dramatically improve the performance of highly selective point or range queries and joins, index selection is the main focus of design advisors. Moreover, design advisors consider materialized views and partitioning as additional optimization goals that are out of scope for our

Storage Adaptivity approach, because materialized views are logical access paths and partitioning is subject of *Data Placement Adaptivity*. Hence, we will focus on related work that addresses solely the index selection problem.

Index selection is an NP-complete problem [35]. It is not feasible to create indexes on all possible column combinations, because of the immense storage and maintenance overhead, which occurs due to updates. Hence, the benefit as well as the costs for maintaining an index need to be evaluated for the current workload. DBDSGN [42] was the first actual design advisor tool build for System R. This tool uses a so-called what-if interface [33] to insinuate the query optimizer hypothetical index configurations. With the help of the what-if interface the design advisor is able to obtain the estimated costs of queries for different index configurations. Nevertheless, since optimizer calls are costly, the number of index configurations the query costs are evaluated for needs to be reduced. One approach are atomic configurations, which consider only one index per table. Additional ways for reducing the number of what-if calls are pruning approaches. For instance, columns that are not referenced by a query do not need to be considered for an index. All of the aforementioned techniques are the foundation for modern design advisor tools. Other index selection-related design advisor techniques consider multi-column indexes [32], design refinement [30] or relaxation [29], and a tighter integration into the query optimizer [139].

Our *Storage Adaptivity-Specific ECL* that triggers adaptations in the *1-Storage* faces a similar index selection problem. However, the usage of the discussed techniques are not suitable for our purposes because of two reasons. First, to do runtime adaptations, the discussed heavyweight approaches are too costly and thus, we use lightweight decision models that are relaxing the goal of finding the optimal solution. Second, the *Living Partition* architecture uses two completely separated optimizers. (1) A global optimizer that generates macro query execution plans (cf., Section 3.4.3) and (2) fast living partition-local optimizers that generate micro query execution plans (cf., Section 5.2.5) based on the actual physical storage layout. Due to this decomposition, none of the optimizers knows the complete query execution plan, which results in a much smaller search space per optimizer.

5.1.3 Adaptive Storages

In contrast to design advisors, adaptive storages autonomously do an online physical storage layout adaptation. The actual adaptation either happens as a side product of the query processing or is incrementally done in background. We split the related work in two major parts. (1) Adaptive indexing approaches and (2) adaptive row-column stores for hybrid transaction-analytical processing (HTAP) workloads.

The first adaptive indexing approach was proposed for the columnar storage model that is organized in binary association tables (BAT) as it is done in MonetDB [21, 23] and is known as database cracking [63, 74]. The idea of database cracking is to incrementally sort and index previously unsorted columns in a query-driven fashion. Every time a query executes a range scan over a column, the range scan predicates are

Table 5.1: Comparison of adaptive storage systems.

Feature	Cracking	H_2O	Tiles	1-Storage
Adaptive Indexes	✓			✓
Partial Indexes	✓			✓
Adaptive NSM		○	○	✓
Adaptive DSM	○	✓	✓	✓
Extensible Storage Modules				✓
Flexible Schema				✓
Unified Storage Interface			✓	✓
Incremental Adaptation	✓		✓	✓
Layout Inhomogeneity			✓	✓
Redundancy Beyond Indexes		✓		✓

used as partitioning predicates and the partitioning of the column is getting more and more fine-grained. Additionally, a cracker index that keeps track of the partitioning is created and maintained. The cracker index starts as a partial index that converges against a full index. A variation of the basic database cracking approach is sideways cracking [64], which propagates the partitioning of the cracker column to a set of additional columns to speed up the tuple reconstruction process. Another variation is stochastic cracking [54]. Because the indexing quality of database cracking depends on the value ranges that are queried, stochastic cracking adds partially arbitrary decisions to the cracking process to make the indexing more robust to outliers.

The counterpart of database cracking for the row-wise storage model is adaptive merging [47]. Here, the starting point is a partitioned B-Tree [46], which is a set of partitions that are sorted internally, but contain overlapping keys. Every time a query searches for specific keys, all partitions need to be scanned and the found key-value pairs are inserted into a new initially clean partition that converges against a full index. Since database cracking and adaptive merging share the use of partitions, but split respectively merge them, hybrid approaches [65] were researched in the context of the columnar DBMS MonetDB. The hybrid approaches turned out to exhibit different characteristics in terms of initial overhead for the first query, convergence speed, and overhead.

Database cracking, adaptive merging, and hybrid versions are promising directions for building an adaptive storage. However, besides being limited to the index selection problem only, all methods face a variety of drawbacks. For instance, they do not consider the trade-off between maintenance costs and query performance benefit, the indexing process can not be incrementally reversed, and the first query faces a high initial overhead. In contrast, our *1-Storage* approach overcomes those drawbacks and is not solely limited to indexing as summarized in Table 5.1.

The next category of adaptive storage systems are hybrid stores that switch between a row-wise and columnar data organization respectively mixes of both at runtime to speed up OLAP or even hybrid transaction-analytical workloads (HTAP). The first system proposed was H_2O [11] that aims at analytical workloads. The H_2O storage model supports the traditional row-wise format, the columnar format, as well as column groups and decides solely based on the workload which physical storage format is used for serving the query. Since H_2O aims at read-intensive workloads, the system keeps multiple data layout versions, which are built as a side product of the query processing to avoid additional data reads for the adaptation process. While pure row-wise or columnar representations are fixed in their configuration, H_2O uses column groups to cluster columns that are accessed often in combination similar to HYRISE. Moreover, the storage manager uses code templates to generate and compile storage format-specific operators on-the-fly.

Another approach is taken by the Peloton DBMS that is based on tiles [13] and aims at HTAP workloads. Peloton horizontally partitions relations into tile groups, which are independently vertically split into tiles. Hence, the system is basically a column group-based storage that is horizontally partitioned. In the living partitions-based environment 1-Storage is supposed to operate in, horizontal partitioning is already given. The actual adaptation process happens per tile group and is executed in background. Moreover, Peloton uses logical tiles as indirection layer to eliminate the need for multiple query processing engines.

In Table 5.1, we give a comparison of our *1-Storage* approach to database cracking, H_2O, and tiles. The first main difference is that none of the mentioned approaches supports adaptive indexing and an adaptive hybrid storage in combination unlike 1-Storage. Moreover, none of the approaches considers a flexible schema or extensible storage modules. Another observation is that neither H_2O nor the tiles-based approach can deal with attributes of a variable length, which is one of the strengths of a row store that does not need any auxiliary data structures such a dictionaries to cope with this issue. Hence, 1-Storage is a holistic approach that is far more than just a combination of adaptive indexing and an adaptive hybrid store. It is able to bring the row, column, and index-oriented worlds together, while being incrementally adaptive, schema flexible, and extensible.

5.2 1-Storage

In this section, we present the concepts of our *1-Storage* approach, which is instantiated for each living partition and actually brings a partition to life due to its highly adaptive nature. The 1-Storage aims at adapting the physical storage layout at runtime to specialize the partition for the current data and workload characteristics the living partition is facing with the high-level goal of saving energy by increasing the performance per power unit. The *Storage ECL*, which is also instantiated per living partition, actually controls the adaptation process and gets coordinated by the other ECLs on the upper hierarchy levels, because a full relation is horizontally split into

many living partitions that should not be adapted simultaneously to obey the query latency limit. Hence, the implicit partitioning of the living partitions architecture is the key enabler for a fine-grained incremental adaptation.

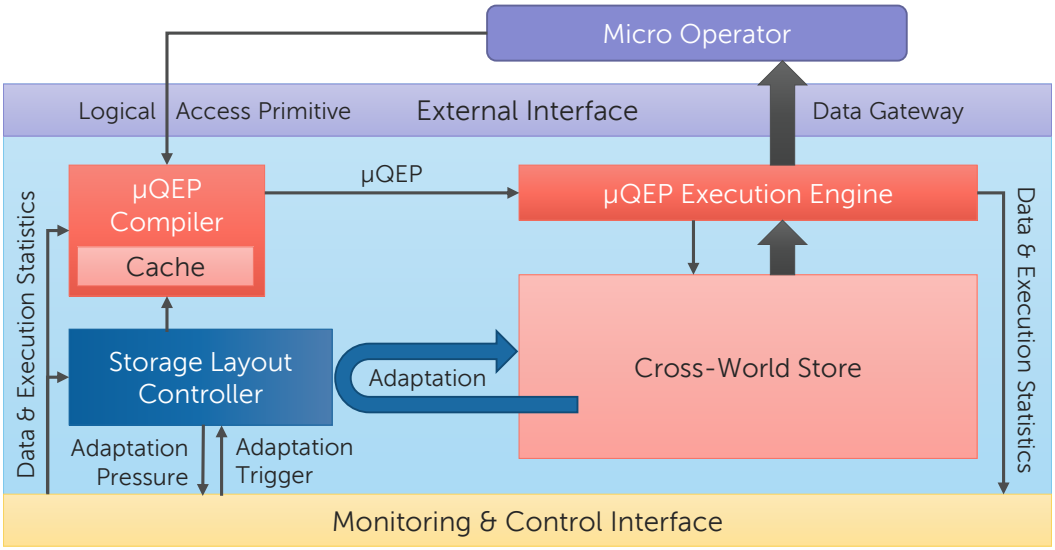


Figure 5.2: 1-Storage component interaction overview.

Figure 5.2 depicts an overview of the 1-Storage architecture including the individual components and their interaction. The objects that are triggering a data access are *Micro Operators*, which are part of a *Dataflow* respectively *Macro Query Execution Plan* (cf., Section 3.4.3). A micro operator requests or manipulates data using a set of predefined logical access primitives and the corresponding parameters. Based on the request, the *μQEP Compiler* uses existing data statistics and current storage layout information to generate a *Micro Query Execution Plan (μQEP)*, which is cached as long as the physical storage layout was not changed and frequent requests of this type arrive. Afterwards, the μQEP is executed by the *μQEP Execution Engine* that operates on the *Cross-World Store*, which actually contains the data. Finally, the requested data is handed over to the micro operator via the *Data Gateway* and data as well as execution statistics are fed back to the monitoring component. The *Storage Layout Controller* periodically checks those statistics and determines whether a storage layout adaptation is necessary or not. If an adaptation is required this *Adaptation Pressure* is reported to the storage adaptivity-specific ECL, which triggers the actual adaptation at a suitable point in time. In the following, we will discuss the individual components of the 1-Storage except for the storage layout controller, which is mostly a part of the storage ECL.

5.2.1 Logical Access Primitives

The 1-Storage offers a small but powerful set of logical access primitives that define in which data the micro operator is interested in respectively how data should be manipulated. In Table 5.2, we list the five access primitives that are powerful enough to process relation queries. Note that some of the access primitives are bound to specific conditions regarding the partitioning of the relation respectively the execution order on the individual partitions. In the following, we will discuss the individual logical access primitives in detail including their parameters and purpose.

Table 5.2: Available logical access primitives in 1-Storage.

Logical Access Primitive	Explicit Partitioning	Partition-Wise Execution	Description
`Insert`			Insertion of a record
`UnorderedScan`			Scan over all partitions in no explicit order
`Lookup`			Specialization of the `UnorderedScan` for lookups
`ConditionalInsert`	✓		Insertion or update of a specific record
`OrderedScan`	✓	✓	Scan over all partitions ordered by an attribute set

`Insert.` This logical access primitive inserts a set of records into the 1-Storage. The parameters are interpreted records (or dynamic records), because 1-Storage is designed to fully support schema flexibility. Each relation maintains a global vector of known attributes including the respective name and data type and a dynamic record includes information about the attributes that are actually defined by the record. Hence, 1-Storage also knows a special `UNDEFINED` value besides the traditional `NULL` value to deal with a flexible schema.

`UnorderedScan.` The unordered version of the scan is the most fundamental access primitive for reading data. Unordered scans are broad-casted to all living partitions of the target relation and are executable in parallel. The parameters of this logical access primitive are: (1) the set of attributes the micro operator is interested in (*desired attributes*), (2) the set of attributes that is needed for validating the scan predicate (*target attributes*), and (3) the scan predicate that includes the static data bindings. All qualifying tuples are handed over to the micro operator for further processing without any particular order.

`Lookup.` The lookup access primitive is a specialized version of the `UnorderedScan` for point data accesses. Instead of a generic predicate, the lookup uses only a list of values and the target attributes need to be equal to (e.g.,

target attribute$_1$ = value$_1$ AND target attribute$_2$ = value$_2$). Due to this specialization, the μQEP compilation is faster, messages get smaller, and messages do not need to be broad-casted in case of a compatible partitioning scheme. This access primitive is used e.g., for point queries or equi-joins.

`ConditionalInsert.` The conditional insert is some kind of an upsert that is primarily used for grouped aggregations also known as reduce function. Therefore, this logical access primitive needs a set of keys and a set of values as parameters, which are encoded as records within the message. 1-Storage checks whether a record with the same key combination is already present. If such a record is not found the keys and values are inserted as records, otherwise the micro operator receives the found values and calculates the new values according to its aggregation function and updates the old values. This access primitive is derived from the `Lookup` and hence, from the `UnroderedScan`. The key attributes are the *target attributes* and the value attributes are the *desired attributes*. To ensure that the same key combinations end up in the same living partition, the key attributes need to be compatible with the partitioning scheme of the relation as a prerequisite.

`OrderedScan` The ordered scan is another modification of the unordered scan, which takes the same parameters and an additional set of *ordering attributes*, but guarantees that a micro operator sees records in the requested order. Hence, the partitioning scheme of the relation needs to be compatible with the *ordering attributes* and the micro operator is executed serially for each living partition. This access primitive is necessary for `order by` clauses.

To summarize, the external interface of our 1-Storage offers five logical access primitives. One access primitive is the insert to add records to the store and the other four fetch data for micro operators. The only logical access primitive that allows the manipulation of existing data is the `ConditionalInsert`. Nevertheless, the conditional insert does in-place updates and is thus not ensuring snapshot isolation for concurrent accesses (cf., Section 3.4.5), because it is meant for intermediate results that are modified in a single transactional context. To actually manipulate data in a safe way, the *Data Gateway* offers *delete* and *update* methods. For instance, the records that need to be deleted or updated are fetched via the `UnorderedScan` or the `Lookup` and the micro operator is able to invoke the respective methods of the data gateway to delete or update specific records.

5.2.2 Data Gateway

The *Data Gateway* is a critical component of our *1-Storage* architecture, because data that is stored in a changing physical format needs to be passed to *Micro Operators* in a unified way. Hence, the data gateway is an indirection layer that adds an additional overhead in terms of data access costs. In the remainder of this section, we will discuss the several options for designing this indirection layer and evaluate

the respective overhead for multiple storage layouts. Moreover, we will point out how most of the induced overhead can be eliminated with the help of just-in-time compilation approaches.

Algorithm 4 Indirect access methods in the context of a micro operator.

```
1:  function MicroOperator::execute(dataGateway)
2:      for each recordHandle/offset in batch do
3:          ...
4:          // Virtual method call
5:          Type *value ← dataGateway.getAccessPath(col).getValue(recordHandle)
6:          ...
7:          // Data Pointer buffer
8:          Type *value ← dataGateway.buffer[col][offset]
9:          ...
10:     end for
11: end function
```

Our 1-Storage processes parametrized logical access primitives and passes data batches to the `Execute` method of the requesting micro operator, which does the actual data processing. Algorithm 4 shows schematically how this method internally operates. The `Execute` method loops over the data batch (line 2–10), which contains record handles (the absolute object identifier) or sequential offsets within the batch based on the indirect access method. The first indirect access method we consider works via virtual function calls (line 5). This method fetches the `AccessPath` object (e.g., a row or column store) that contains a respective attribute and calls its virtual `getValue` method using the record handle as parameter to fetch the actual pointer to the value of the attribute. 1-Storage uses data pointers, because it mostly operates on untyped data and it is up to the operator to interpret the binary data. While this indirect access method does the actual indirect via virtual function calls, our second option uses data pointer buffers (line 8) to do the indirection. In the data pointer buffer-based approach, 1-Storage reconstructs the *desired attributes* of a record in a buffer, which is used by the micro operator to fetch the data pointers using the offset within the batch. The size of the data buffer is $\text{size}_{\text{batch}} \cdot |\text{desired attributes}| \cdot \text{size}_{\text{pointer}}$ and it is organized in a column-first memory layout.

We conducted a series of experiments to evaluate the virtual function call-based and the data pointer buffer-based indirect access method for different physical storage layouts. In the following, we will refer to both methods as *virtual* and *buffer* method. For our experimental setup we use five different non-hybrid storage layouts. A column store that has a high scan performance (Column), a row store that uses a static schema (Row Static), and a row store that uses a flexible schema (Row Dynamic). Both row store variants either contain a single attribute (Small) or additional 8 dummy attributes od the same size (Big). To put the data gateway under stress, our workload does an aggregation (sum) over a single attribute.

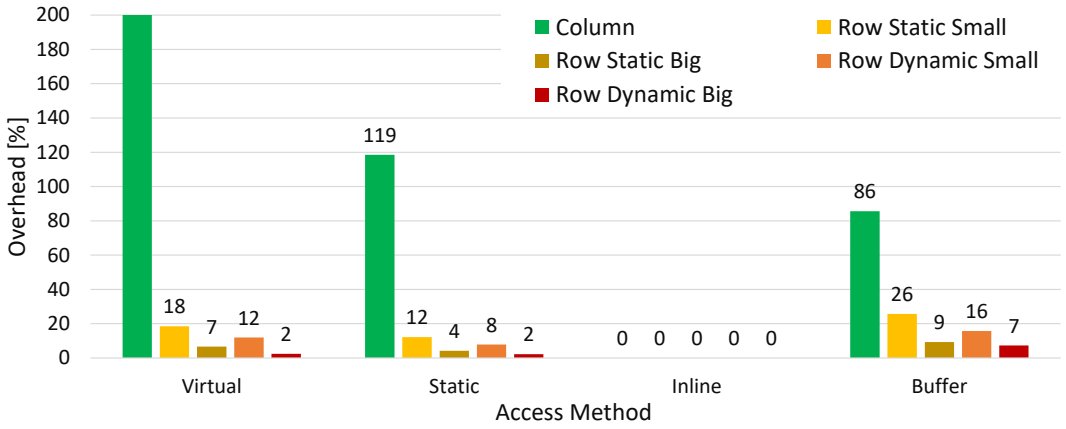


Figure 5.3: Overhead of indirect access methods for different storage formats (8 Bytes fields).

In Figure 5.3, we visualized the measured overhead for the five physical storage layouts and the respective indirect access method relative to a direct access as 0 % overhead baseline. Besides the virtual access method, we also added a modification where the class of the `AccessPath` object is known before hand, which results in a static function call (Static) and a modification where the `GetValue` method was additionally inlined by the compiler (Inline). Both modified access methods can not be used as indirect access method, but help to understand the source of the overhead induced by the virtual method. The data size of the raw data was set to 1 GB, which is enough to be memory-bound, and the size of the data fields is 8 Bytes. As the overhead measurements show, the column store experiences the most overhead for the virtual as well as the buffer method, because of its superior scan performance and thus, the relative indirection overhead amounts to a high fraction of the overall execution time. For the slower row store versions, which are not designed for such a workload, we observe much less overhead of the indirect data access. A comparison of the results for the virtual and for the buffer method show that the buffer method adds less than half of the overhead the virtual method does in the column store case. In the row store cases, we see the contrary behavior, but with a much smaller relative difference.

To understand the high overhead of the virtual method, we focus on the column store case. The measurements for the static function call adds only about half of the overhead compared to the virtual function call and the inlined version adds no overhead at all, because the same code is generated as for the direct access. This leads to the conclusion that the virtual function call is responsible for half of the overhead and the remaining overhead accounts for optimizations that could not be made by the compiler such as loop vectorization. Moreover, non-inlined function calls add additional instructions for the parameter passing and the general function

preamble (e.g., saving the stack pointer). In contrast, operator code that works on data buffers can be vectorized and induces no costs for high amount of function calls.

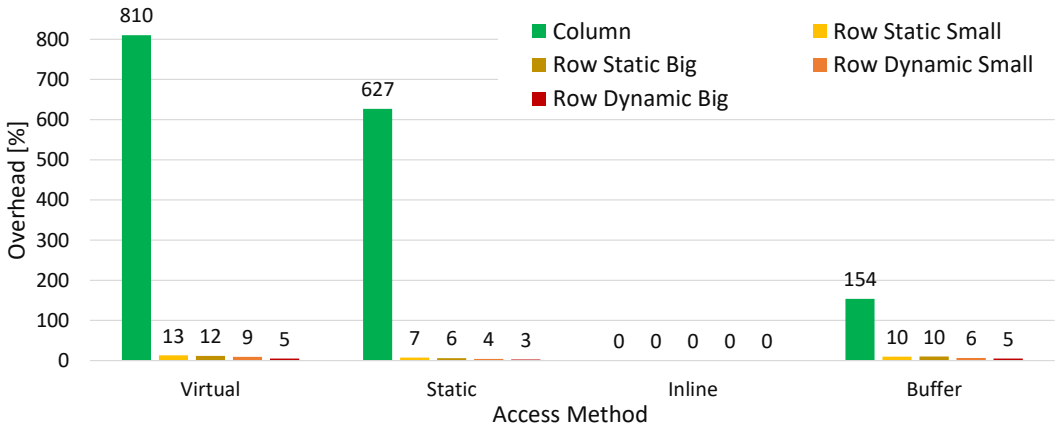


Figure 5.4: Overhead of indirect access methods for different storage formats (1 Byte fields).

In Figure 5.4, we repeated the same experiment with a data field size of 1 Byte, which effectively leads to less cache references per time unit and thus, even more pressure is put on the indirection layer. The overall raw data size is still 1 GB. As the measurements show, the overhead of the virtual method increases by a factor of 3, while the data buffer method overhead only doubles compared to a bigger data field size. Moreover, the static method only saves a quarter of the virtual method overhead. Regarding the row store data layouts, we see a decrease of the overhead compared to the previous experiment.

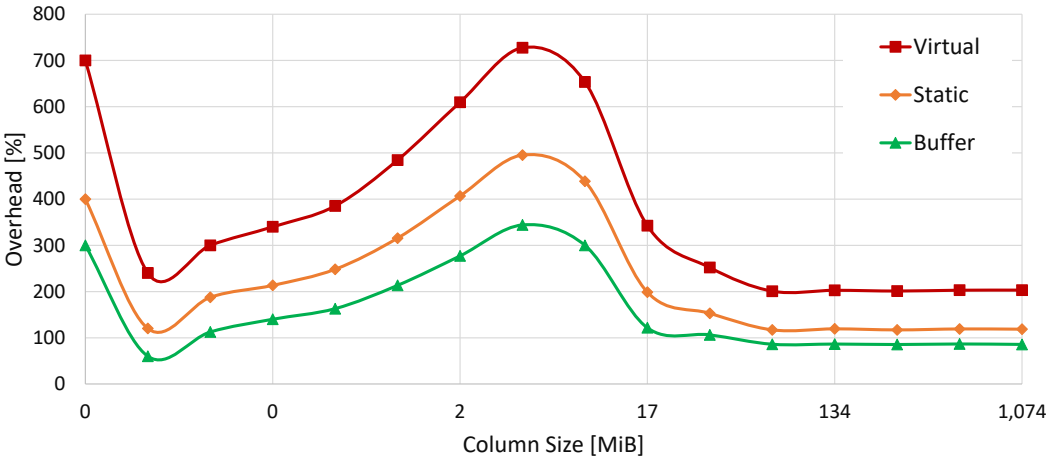


Figure 5.5: Overhead of indirect access methods for different column sizes.

In Figure 5.5, we investigated the dependency of the raw data size on the overhead of the indirect access methods for the column store case with 8 Bytes data fields. As the measurements show, the relative overhead changes while the data fits into the caches of the CPU and is getting constant in the memory-bound case. This fluctuations happen, because the absolute performance of the column store changes depending on the cache level it fits into. We also observe that the buffer method always performs better compared to the virtual method.

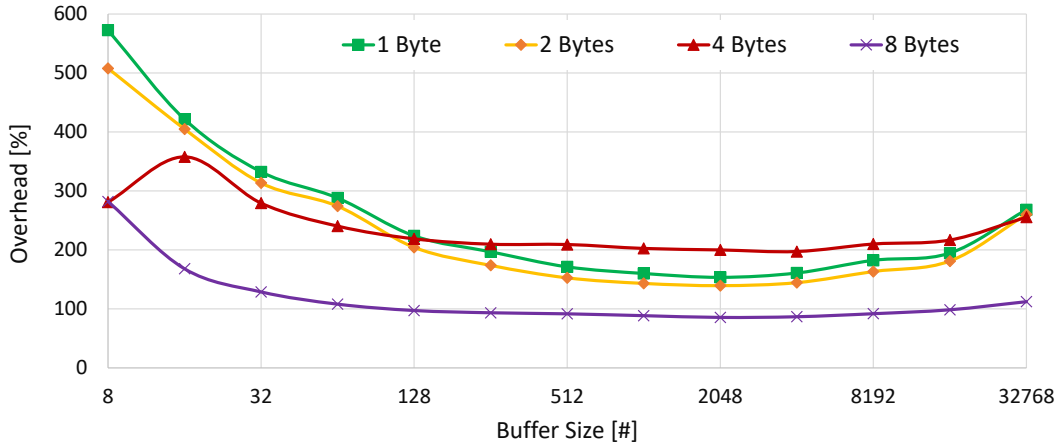


Figure 5.6: Overhead of indirect access for different buffer sizes and field sizes.

Since the buffer method turned out to be the best choice in the previous experiments, we investigate the data buffer size parameter of this method in our last experiment. Once again, we use a column store with 1 GB raw data size and conducted the measurements for 1, 2, 4, and 8 Bytes data fields. In Figure 5.6, we visualized the respective results, which show that a buffer of 2048 elements gives us the least overhead. Nevertheless, this experiment only covers the case that a single column is scanned. In practice, a data buffer of 512 elements turned out to be the best choice when scanning over multiple columns.

To Summarize, our investigation of the virtual and buffer method revealed that the buffer method has the least overhead in a wide majority of data and data layout scenarios. However, the buffer method still adds a significant overhead, which is the price that needs to be paid to have a fully adaptive storage. In the future work, we want to explore how just-in-time compilation-based approaches [11, 56] can be applied for the data gateway, because our experiments showed that additional compiler knowledge that is only available at runtime can fully eliminate the overhead. Nevertheless, just-in-time compilation adds costs for the compilation process itself and thus, a generic indirection layer usually reduces the total costs for small ad-hoc queries compared to a compilation-based approach. Moreover, we want to investigate how the data gateway can be extended to allow operators to directly operate on compressed or encrypted storage layouts.

5.2.3 Cross-World Store

In this section, we present the heart of our *1-Storage* approach, which is the *Cross-World Store*. The cross-world store is able to store records in a wide variety of physical storage layouts and supports features like redundancy (beyond indexes), schema flexibility, and extensibility using custom storage modules. We will discuss the extensibility feature in Section 5.2.4 in detail and present a set of exemplary storage module implementations and focus on the overall concept of the cross-world store in this section. Another core feature of the cross-world store is its ability to adapt its physical storage layout at runtime, which will be the subject of Section 5.3.2.

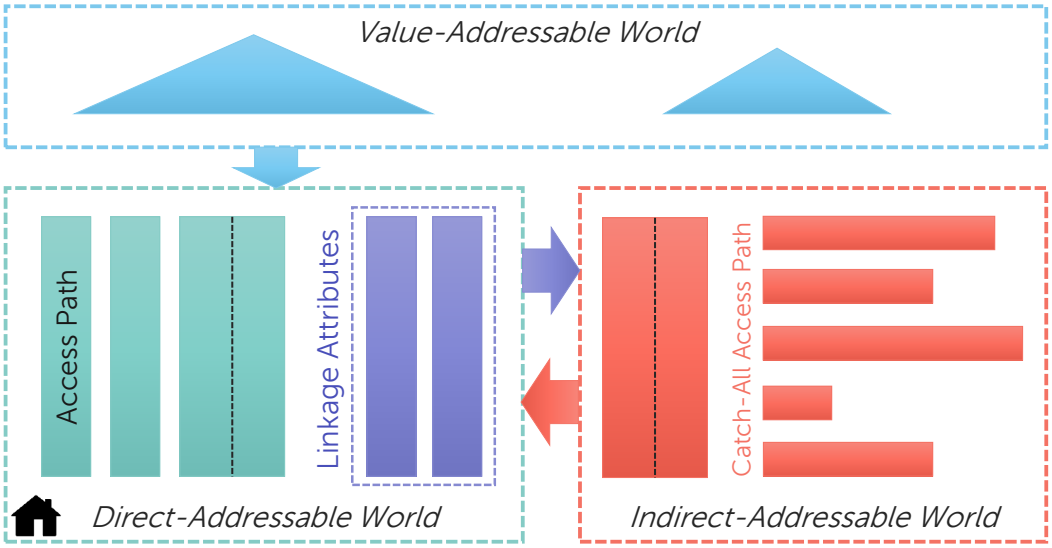


Figure 5.7: Overview of the Cross-World Store.

Figure 5.7 gives an overview of the internal organization of the cross-world store. As shown, the cross-world store knows three worlds an access path is supposed to operate in. An access path is an instance of a storage module that physically stores and organizes the values of a set of attributes, which we call *partial records*. The difference between the individual worlds is the addressing mode that they are using. In the following, we will discuss the three worlds in detail.

Direct-Addressable World. In this world, all partial records of a full record placed in the direct-addressable world have the same address similar to a positional addressed column store. Hence, the address is either equal to the insert order or the ordering of a single access path of this world and the different access paths cannot be ordered individually. The direct-addressable world is suitable for access paths that need no indirection for finding a partial record and preserve their order such as basic column stores and column groups that do not contain attributes of a variable length. Furthermore, the direct-addressable

world is the home world of the cross-world store, because it is heavily involved in the record reconstruction process.

Indirect-Addressable World. The indirect-addressable world contains those access paths that need an indirection for being reconstructed. This reconstruction happens with the help of automatically generated *Linkage Attributes*, which are stored in the direct-addressable world and perform the actual indirection. Moreover, access paths of the indirect-addressable world can contain implicit *backlink* attributes to link partial records to their respective direct address of the direct-addressable world. However, the presence of the backlink attribute in an access path of this world depends on the workload and the choices made by the *Storage Layout Controller*. The indirect-addressable world is suitable for access paths like static or dynamic row stores that do not preserve a particular order of the partial records or contain attributes of a variable length, or binary association tables (BATs). Due to the flexibility of the indirect-addressable world, it contains a special *catch-all access path*, which is needed for schema flexibility and stores values of attributes that were recently added or are not mapped to any other access paths.

Value-Addressable World. In this world, a partial record is addressed using a specific value or value ranges. Hence, the value-addressable world is suitable for storing primary or secondary indexes, which are auxiliary data structures that speed up the selection of records. If an index stores a partial record whose attributes are not contained in any other access path it is a primary index. In contrast, if the index contains redundant attributes or no attributes at all it is a secondary index. Additionally, indexes can contain the direct address of a record to link the partial record to the direct-addressable home world, which is the same special *backlink* attribute as it is employed by the indirect-addressable world.

To summarize, the cross-world store uses generic access paths that are placed in one of the three aforementioned worlds depending on their characteristics and addressing mode. The cross-world store is responsible for maintaining auxiliary attributes (i.e., linkage and backlink attributes) to enable the reconstruction of partial records that exist in the different worlds. Nevertheless, since this reconstruction across multiple world imposes additional costs for the indirection, the *Storage Layout Controller* tries to cluster attributes that are accessed in combination in the same world or even the same access path. We will give examples for access paths and storage layout configurations in the following sections of this chapter. Moreover, the cross-world store maintains a special *catch-all access path* in the indirect-addressable world as a reception center for recently added attributes in case of a flexible schema.

5.2.4 Storage Modules

The cross-world store is not limited to a specific set of access path implementations. Instead, it uses *Storage Modules* that implement a generic interface, which enables the cross-world store to be extended by any kind of access path that fits into one of our three worlds. Each storage module exposes a set of properties to define for which scenarios it is applicable and a storage module is additionally benchmarked by the *1-Storage* to obtain its performance characteristics (cf., Section 5.3.1) and to feed the internal cost model of the *Storage Layout Controller* (cf., Section 5.3.2), which does the actual physical storage layout adaptation.

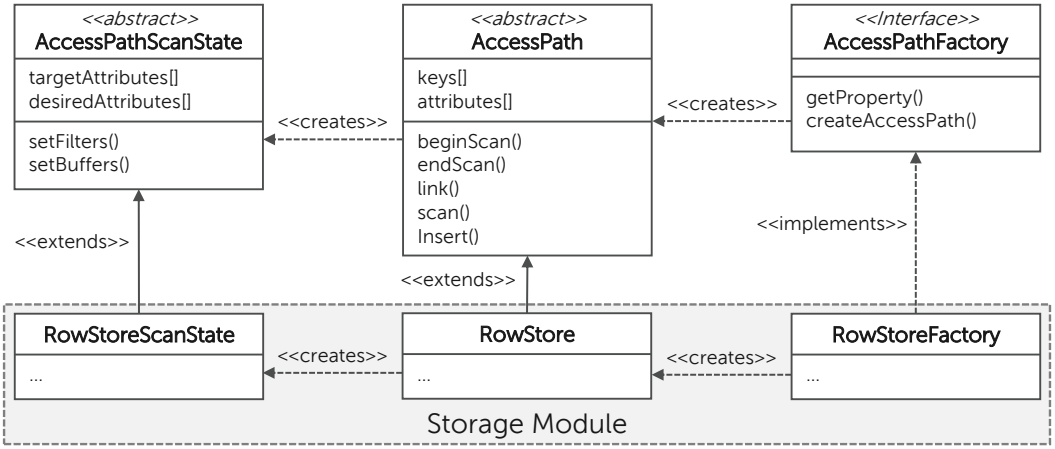


Figure 5.8: Interfaces and class hierarchy of storage modules.

A storage module consists of a set of classes that implement or extend the interfaces respectively abstract classes of the cross-world store. In Figure 5.8, we visualized the simplified class hierarchy of the cross-world store and an exemplary storage module in the UML notation. As shown, the interfaces comprises the `AccessPathFactory`, the `AccessPath`, and the `AccessPathScanState`. In the following, we will describe the three parts of the interface in more detail.

`AccessPathFactory.` This interface follows the factory software design pattern and gives 1-Storage information about the implemented access path and allows to create instances of it. An `AccessPathFactory` needs to implement two major methods, which is the `getProperty` method and the `createAccessPath` method. The `getProperty` method gives information about the configuration space of the access path and tells 1-Storage to which world it belongs to. In Table 5.3, we enumerate all of the properties and their valid values that are queried via this method. The `createAccessPath` method returns an instance of the implemented access path and takes a set of parameters that define how this access path object is configured,e.g., which attributes it contains.

Table 5.3: Properties of an access path.

Property	Possible Values	Description
`Addressing Mode`	DIRECT, INDIRECT, VALUE	addressing mode used to address partial records in the access path
`Attribute Count`	0 .. *, ANY, DYNAMIC	Number of attributes the access path can contain
`Key Count`	0 .. *	Number of attributes the access path can index
`NULL Values`	FALSE, TRUE	NULL values supported
`UNDEFINED Values`	FALSE, TRUE	UNDEFINED values supported
`Variable Length Attributes`	FALSE, TRUE	Attributes of a variable length are supported

`AccessPath.` This abstract class is derived to implement the actual access path that contains the partial records respectively implements the additional indexing data structure. The object contains the configuration parameters as attributes and the three core methods `insert`, `scan`, and `link`. The `insert` method takes a dynamic record and converts it into its internal storage format. The `scan` method scans sequentially over the data. In case of a directly addressed storage module, the scan order needs to be equal to the address order. In any other cases the scan order is arbitrary. The `link` method receives a batch of addresses and mainly joins a set of requested attributes to this batch. While the `insert` method is stateless, the `scan` and `link` methods require an access path-specific state that is obtained via the `beginScan` method, which returns an `AccessPathScanState` object.

`AccessPathScanState.` The `AccessPathScanState` is an auxiliary object, which keeps the state of a `scan` or `link` operation of an `AccessPath`. This state stores information about the attributes the respective operation is interested in and whether filters on some attributes should be applied on the access path-level. Moreover, the `AccessPathScanState` is parametrized to know where the result of the operation should be written to in the data buffers of the *Data Gateway*. Besides this information, the internal position within a `scan` operation is stored here, because a scan happens in batches and thus, the `scan` method is likely to be called multiple times for a single scan operation. Note that index scans use exactly the same interface.

To summarize, our 1-Storage uses a generic interface for storage modules that implement the actual access paths. Hence, a physical storage layout is a composition of access path instances and their respective configurations. To actually query or manipulate partial records that are stored in the individual access paths, the

μQEP Execution Engine uses the internal access primitives `insert`, `scan`, and `link`. Moreover, a storage module supplies the *Storage Layout Controller* with additional information about the feature set that is implemented by its access path. This meta knowledge is considered for the storage layout adaptation process.

Table 5.4: List of exemplary storage modules used for evaluation. The highlighted properties qualify the dynamic row store for the catch-all access path

Storage Module	Addr. Mode	Attr. Count	Key Count	NULL Values	UNDEF. Values	Var. Attr.
Row Store Static	INDIRECT	ANY	0	✓	✓	✓
Row Store Dynamic	INDIRECT	DYNAMIC	0	✓	✓	✓
Column Group Store	DIRECT	ANY	0	✓	✓	
Single Column Store	DIRECT	1	0			
RB-Tree	VALUE	0	1	✓	✓	✓

For further examples and evaluations of our 1-Storage, we use a set of storage modules that covers a wide range of properties and performance characteristics. Table 5.4 lists those five storage modules as well as their properties. We use two row store implementations that are designed to reside in the indirect-addressable world. Both row store implementations organize their partial records in linked pages and the pointer to a partial record is used as address. While the static version of the row store uses a fixed record layout, the dynamic version stores data in dynamic records that contain additional schema information. Because of its properties (highlighted in the table), the dynamic row store is selected by 1-Storage as the *catch-all access path*. For the direct-addressable world, we use a column group and a column storage module. While the column group implementation supports multiple attributes as well as `NULL` and `UNDEFINED` values, the column store implementation is limited to a single attribute and defined non-`NULL` values. Both implementations lack support for attributes of a variable size, which is a consequence of the direct addressing mode. Finally, we use an RB-Tree [15] implementation for the value-addressable world, which supports a single key attribute and no additional attributes. Hence, this data structure plays the role of a secondary index.

5.2.5 Micro QEP Compiler and Execution Engine

In this section, we discuss how micro query execution plans (μQEP) are compiled and executed. In its essence, a μQEP is an ordered list of parametrized `insert`, `scan`, and `link` operations that are executed on the respective `AccessPath` objects. As discussed in Section 5.2.1, the 1-Storage knows two fundamental logical access primitives which are the `Insert` and `UnorderedScan`. In the following, we will describe

how the μQEPs for both logical access primitives are compiled and executed using the exemplary physical storage layout visualized in Figure 5.9. This storage layout consists of three single column store instance, each containing one attribute that can not be `NULL` or `UNDEFINED`. One of those attributes is the MVCC attribute, which is a special system attribute that is used to determine the visibility of a record. Moreover, the direct-addressable world contains two internal linkage attributes, which are stored in a column group access path and refer to a static row store (`Name` attribute of variable length) respectively to the catch-all access path (containing the sparse `Phone` attribute), which uses the dynamic row store as implementation. The value-addressable world contains two indexes and one of them is the RB-Tree access path that is indexed on the `Age` attribute.

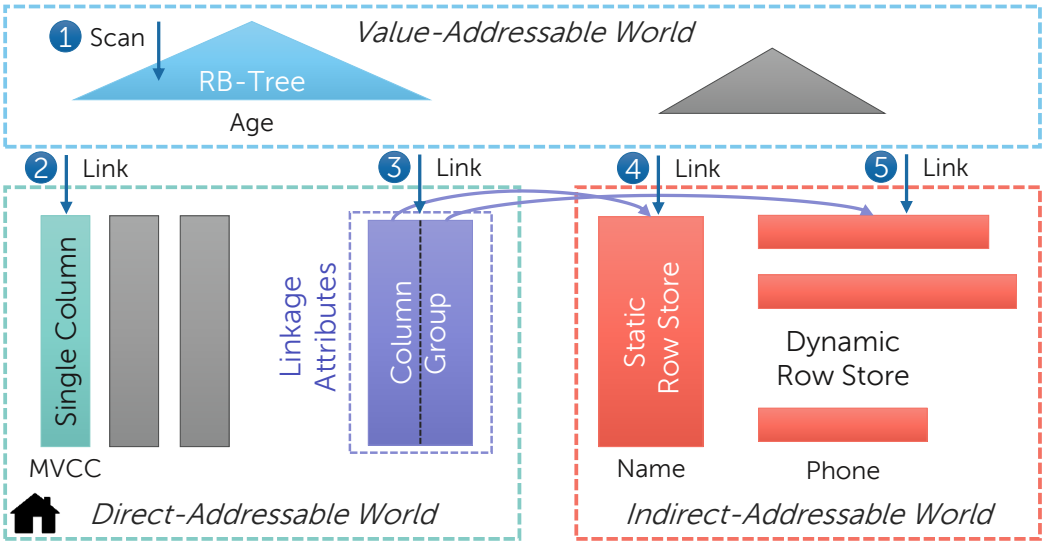


Figure 5.9: Exemplary physical storage layout and micro query execution plan.

Insert. The goal of the `Insert` access primitive is to store a full record in the cross-world store. Hence, the task of the μQEP is to split the full record into multiple partial records, which are inserted into the individual access paths. There are four specialties that need to be considered during this process: (1) attributes can be stored redundantly in multiple access paths, (2) access paths in the indirect-addressable world need to write the partial record address back to the corresponding access path of their linkage attribute (3) attributes can not be mapped to any specific access path and should thus be stored in the catch-all access path, and (4) the local transaction manager needs to keep track of the written MVCC data. Because this is a trivial process of decomposing the record into partial records, we will focus on the more sophisticated `UnorderedScan` scenario.

UnorderedScan. While the `Insert` access primitive splits full records into partial records, the `UnorderedScan` and its specialized versions are responsible for recon-

structing a subset of the full record again and additionally to find the records that qualify for the given predicate. To make this process tangible, we start with an example where the *target attribute* is `Age` and the *desired attributes* are `Name` and `Phone`. A possible μQEP for this parametrization of the `UnorderedScan` is shown in Figure 5.9. In this example, the μQEP comprises five `scan` respectively `link` operations on the access paths. The first operation (1) is a `scan` on the RB-Tree, which parametrizes the `AccessPathScanState` of the RB-Tree to filter the `Age` attribute for the given predicate and return the *record handles*, which are equal to the address used in the direct-addressable world. Since this operation filters the full record set, all of the following operations need to be `link` operations. Hence, the second operation (2) executes a `link` on the column store that contains the MVCC information. The corresponding scan state is configured to link the MVCC attribute the record handles. The successive operation (3) does the same, but links the system attributes needed for the linkage to the access paths in the indirect-addressable world. Finally, `link` operations 4 and 5 (4 and 5) link the `Name` respectively `Phone` attribute of the respective access paths. However, these operations do not use the record handles. Instead, they use the addresses obtained from the linkage attributes in the previous step.

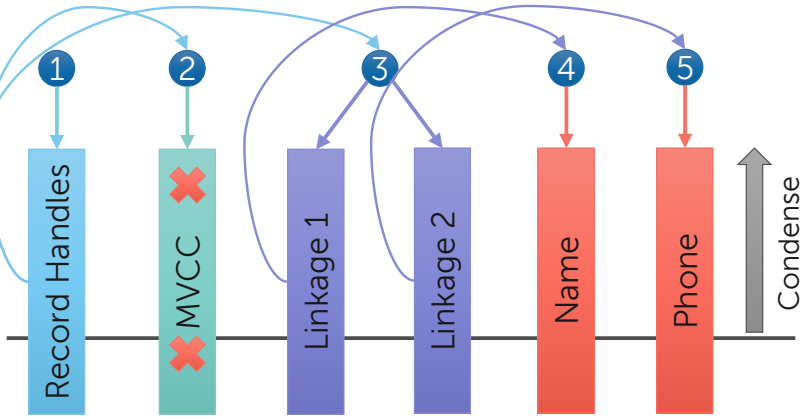


Figure 5.10: Exemplary micro QEP execution and buffer mappings.

Figure 5.10 visualizes the μQEP from the perspective of the *μQEP Execution Engine*, which is responsible for filling the buffers of the *Data Gateway* that are passed in batches to the *Micro Operator*. Note that μQEP are executed single-threaded, because it runs within a single LPV. To execute this μQEP, the execution engine sets up six buffers each of the same batch size. The first operation (1) is configured to fill the record handle buffer, which is always present and contains the direct addresses of the records. Operation 2 and 3 (2 and 3) are parametrized to use this record handle buffer for linking the respective attributes, which are written back to the respective buffers (i.e., MVCC, Linkage 1, and Linkage 2 buffer). While the MVCC buffer contains pointers to the respective MVCC values, both linkage

buffers contain addresses to partial records in the indirect-addressable world. Thus, operation 4 and 5 (4 and 5) are configured to link against those indirect addresses and store the respective attribute value pointers in the `Name` and `Phone` buffer. Afterwards, the execution engine validates the MVCC attributes in the transactional context of the requesting micro operator and marks invisible records followed by a condensation step of the attribute buffers that are effectively passed to the micro operator (i.e., `Name` and `Phone`). If the condensation freed a high fraction of free space in the buffer batch, the execution engine repeats the execution plan. Otherwise, the buffer batch is passed to the micro operator. Since data is passed in batches via the data gateway, access path operations need to preserve their internal state like an iterator. As soon as the last batch is processed by the micro operator, the scan states of the access path operation are freed and execution statistics are send to the monitoring component of the local 1-Storage.

As the example demonstrates, the compilation of a μQEP for an `UnorderedScan` is not trivial, but not as complex as the compilation of a traditional full query execution plan that needs to consider multiple physical operator implementations (e.g., for a join). However, both query compilers share a lot of similarities, but also exhibit differences. For instance, the μQEP compiler needs to be aware of linkage columns to link access paths of different worlds with each other. To generalize the μQEP compilation, we use the following pattern:

(1) Gather *target attribute* statistics regarding their selectivity and decide on a predicate evaluation order. Additionally, mark *target attributes* that are not selective enough to be evaluated via an index.

(2) Search the storage layout for indexes on *target attributes* in the determined evaluation order and add a `scan` operation to the plan for the first index found. All following indexes add a `link` operation. If an index contains additional attributes that are needed for the request, add them to the operation.

(3) Search the direct-addressable world and afterwards the indirect-addressable world for remaining *target attributes* and add a `scan` operation if the plan is empty or a `link` operation if not.

(4) Search the direct-addressable world and afterwards the indirect-addressable world for remaining *desired attributes* and add a `scan` operation if the plan is empty or a `link` operation if not. Prefer access paths that contain combinations of *desired attributes*.

(5) Inject operations to fetch addresses of access paths in the plan residing in the indirect-addressable world to obtain the indirect addresses from the linkage attributes.

We do not claim that this compilation strategy finds the optimal plan and consider it as a heuristics-based approach to find a good plan. Additionally, the μQEP compiler is often limited in its freedom, because it is not allowed to build intermediate

data structures (e.g., a hash table) and thus, the physical storage layout is usually fixed during μQEP compilation and execution. Due to this fixed storage layout assumption, the effective query optimization happens mainly during the storage layout adaptation process. Moreover, the μQEP compilation is much more time-critical, because it needs to happen for every request a 1-Storage instance receives. Hence, 1-Storage tries to cache μQEPs whenever possible.

5.3 Storage Adaptivity-Specific Energy-Control Loop

In this section, we present details about the actual physical storage layout adaptation process, which is mainly controlled by the *Storage Adaptivity-Specific ECL* that is instantiates a *Storage ECL* per *1-Storage* instance and thus, per *Living Partition* as shown in Figure 5.11. The central component of this ECL type is the *Storage Layout Controller*, which considers data and execution statistics generated by the *μQEP Execution Engine*. Moreover, the storage layout controller uses properties and performance statistics of the *Storage Modules* that are registered in the global *Storage Module Manager*. While storage module properties are reported via the respective interface (cf., Section 5.2.4), the performance statistics are obtained via an initial benchmarking process. Using all of this information, the storage layout controller periodically computes the target storage layout and checks for necessary changes to the current storage layout. The actual decision on the time point for the adaptation is subject of the ECLs in the upper hierarchy.

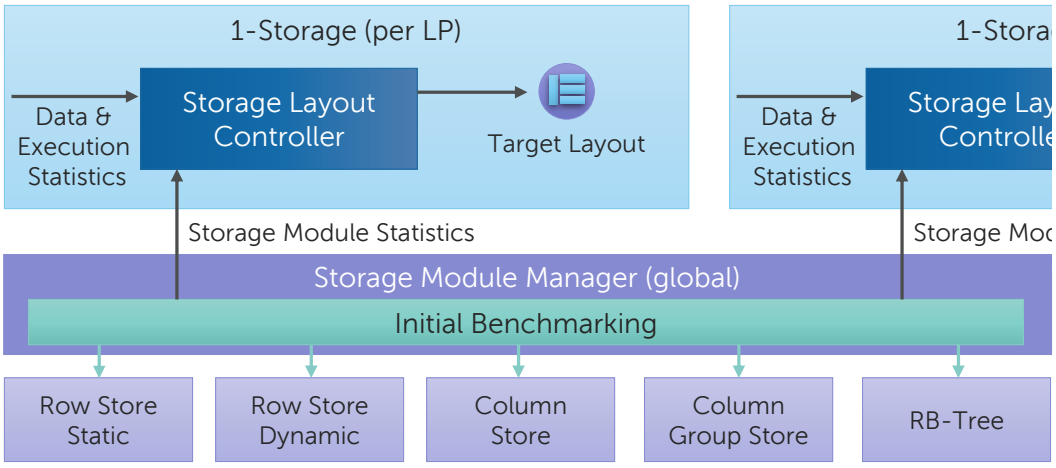


Figure 5.11: Overview of the Storage ECL.

The remainder of this section is structured as follows. First, we will discuss the initial storage module benchmarking procedure and present the results for our five example storage module implementations. Afterwards, we present details on the decision making of the storage layout controller. Finally, we show how the storage

adaptivity-specific ECL integrates with the other ECLs in our hierarchy and thus, at which point in time adaptations are triggered.

5.3.1 Storage Module Benchmarking

The aim of the initial storage module benchmarking process is to supply the *Storage Layout Controller* with additional knowledge about the performance characteristics of the registered *Storage Modules*. Those characteristics are not known beforehand, because 1-Storage supports arbitrary storage modules that have an unknown implementation. Moreover, the performance of an implementation depends on the hardware it is running on. The benchmarking procedure mainly evaluates the performance of the three access path primitives, which are the `insert`, `scan`, and `link` for different data characteristics. The evaluation of the `scan` primitive is split into a scan over the entire data and a highly selective `scan` using a filter (`lookup`). Besides the performance statistics, the benchmark also captures the memory consumption of an access path. The benchmarking process happens in two stages. In the first stage, storage modules are evaluated using a single attribute and in the second stage, multiple attributes are stored in the individual access paths. In the following, we present the results of our example storage module implementations (cf., Table 5.4) for the first stage.

Stage 1 - Single Attributes

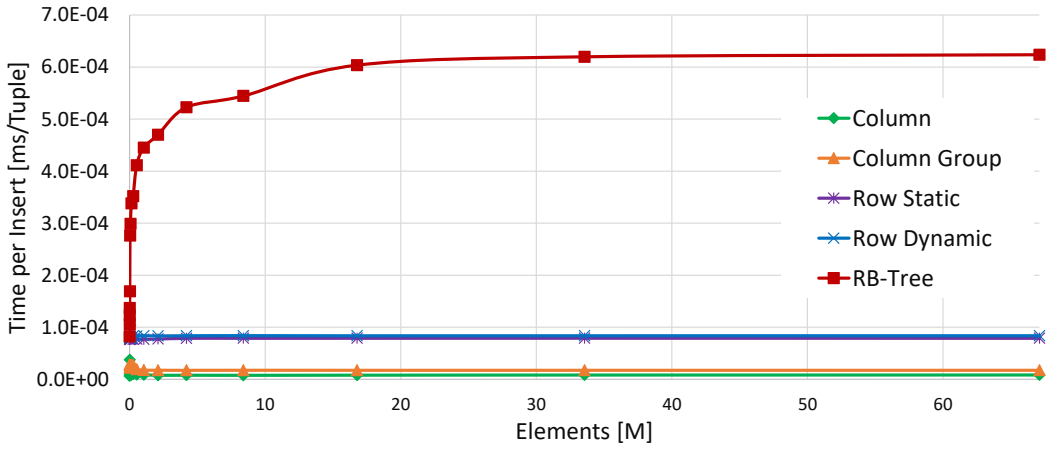


Figure 5.12: Insert costs of different storage modules for element count.

In Figure 5.12, we visualized the benchmark results for the `insert` primitive. The benchmark inserts 4 Bytes integers into the respective access path and captures the insert time per tuple for a different amount of partial records already present in the access path. As the measurements show, the `Column` and the `Column Group` store exhibit the best insert performance followed by the static and dynamic version of

the Row store. While those four access paths show a constant insert time for different sizes of the access path, the insert time of the RB-Tree depends on the size and is significantly higher, because it is a tree-based data structure.

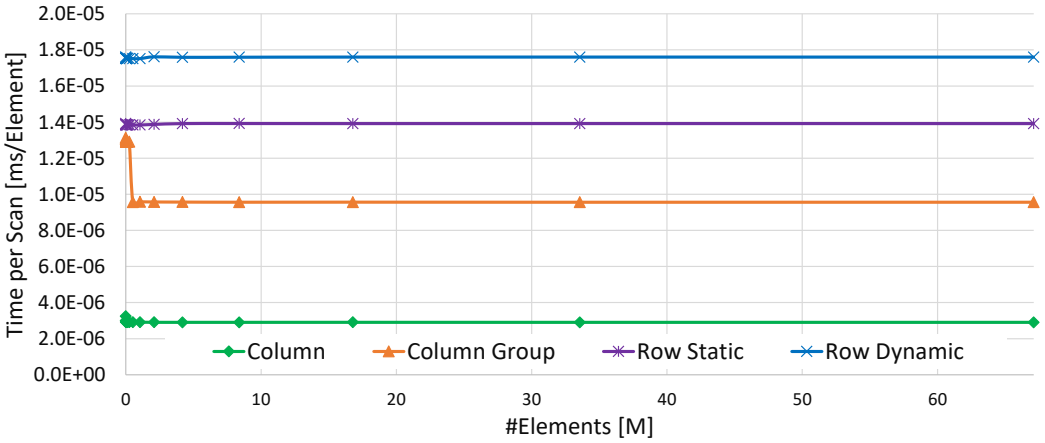


Figure 5.13: Scan costs per element of different storage modules for element count.

In Figure 5.13, we present the results for the same setup, but this time for the scan primitive. The chart does not contain the RB-Tree access path, because 1-Storage never executes full scans on access paths of the value-addressable world. Moreover, the benchmark is designed to work on uncached data structures, because we can not assume cache residency in real-life scenarios and thus, the worst-case scenario is assumed. The benchmark results show a constant time per element during the scan operations for all access path sizes. However, the performance numbers differ per storage module. The best throughput is achieved by the Column store, which organizes the data sequentially without additional meta information and interpretation overhead. The worst throughput was measured for the Dynamic Row store, which needs to scan over a larger memory region and faces a high interpretation overhead. Nevertheless, the Dynamic Row store offers features that are not provided by the simple Column store.

In Figure 5.14, we visualized the benchmark results for the lookup primitive, which is equal to the scan operation with a highly selective filter. The measurements (log scale) show that the RB-Tree performs significantly better compared to the other storage modules, because lookups are the strength of index structures. In contrast, all of the other access paths need to scan sequentially over their entire data.

Finally, Figure 5.15 presents the measurements for the link operation. For this primitive, the benchmark links the containing attribute to a batch of random addresses referencing partial records contained in the access path. Once again, the Column store exhibits the best performance, because it only computes the pointer corresponding to the direct address. In contrast, the Column Group store needs to additionally find the containing page and needs to access the NULL and UNDEFINED bitmap, which is stored at the beginning of the page. Both Row store versions also

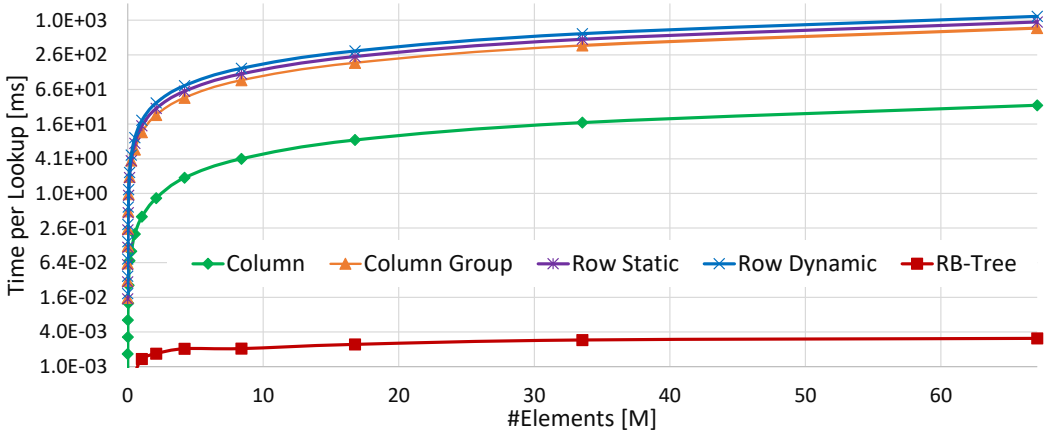


Figure 5.14: Lookup costs of different storage modules for element count.

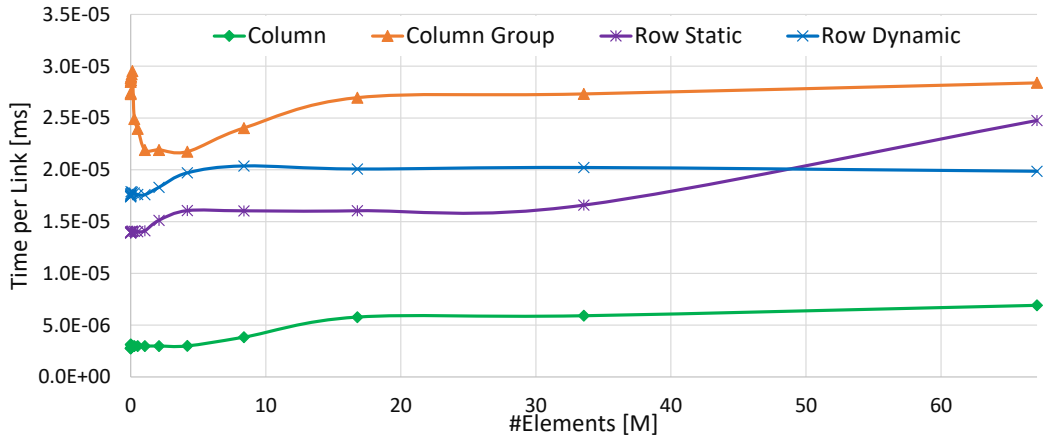


Figure 5.15: Link costs of different storage modules for element count.

need to access those bitmaps, which are stored very close to the linked attribute value resulting in a slightly better performance.

To summarize the results of the first stage, we visualized the benefits of the example storage module implementations in Figure 5.16. The radar chart has the dimensions for the primitives `insert`, `scan`, `lookup`, and `link`. Additionally, the chart contains the memory size of the access paths (less is better). This comparison shows a clear advantage of the `Column` store implementation for all dimensions except for the `lookup`, which is dominated by the `RB-Tree` that performs bad in all other aspects. Compared to the `Column` store, both `Row` store versions exhibit a fair performance, but provide a rich set of features such as multiple variable sized attributes as well as support for `NULL` and `UNDEFINED` values (cf., Table 5.3). The `Column Group` access path takes its place between the `Column` and `Row` store.

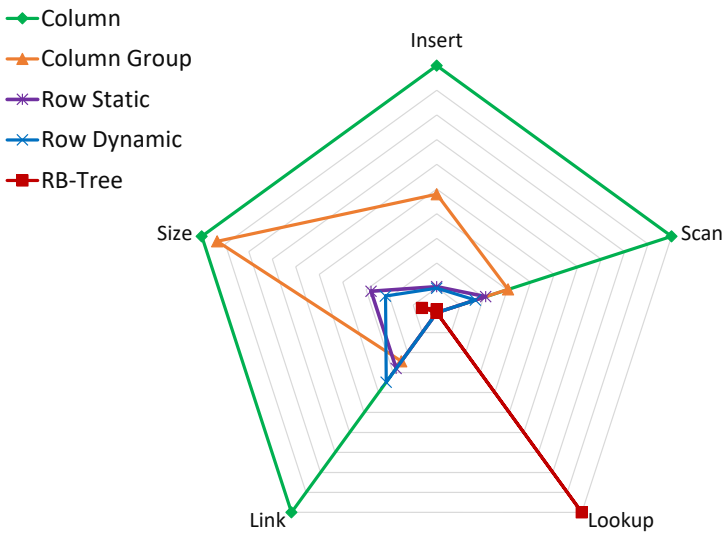


Figure 5.16: Comparison of all storage modules.

Stage 2 - Multiple Attributes

While stage 1 assumes an individual access path for each attribute, stage 2 benchmarks the benefit of grouping multiple attributes into a single access path of the same type. Hence, the benchmark puts a growing number of N attributes into a single access path and compares the performance to the measurements for N individual access paths. A prerequisite for such a grouping is that the respective storage module supports multiple attributes, which is only the case for the `Column Group`, `Static Row`, and `Dynamic Row` store. For demonstration purposes, we also add the `Column` store numbers to our results and calculate the reference performance by multiplying the performance of a single access path instance with N. In the following we present the results for the `insert`, `scan`, and `link` primitive.

In Figure 5.17, we give the results for the `insert` operation. No matter how much attributes are grouped, the isolated or the grouped access paths always contain a total of 64 million elements. For instance, in case of two grouped attributes, the two isolated access paths contain 32 million elements each and the grouped access path contains 64 million elements. Hence, the data sizes are equal to get a fair comparison. As the measurements show, all access paths benefit from a grouping, except for the `Column` store, which performs bad for a high amount attributes, because each attribute is written to a distant memory location. In contrast, all of the other access paths take their advantage from a better cache locality. We also see that `Column Group` store starts to perform worse for a high number of grouped attributes, which is caused by an implementation detail. However, 1-Storage is able to detect this weakness by considering the results of the initial benchmark.

In Figure 5.18 and 5.19, we visualized the results for the `scan` respectively `link` operation. The measurements show that both operations benefit from the attribute

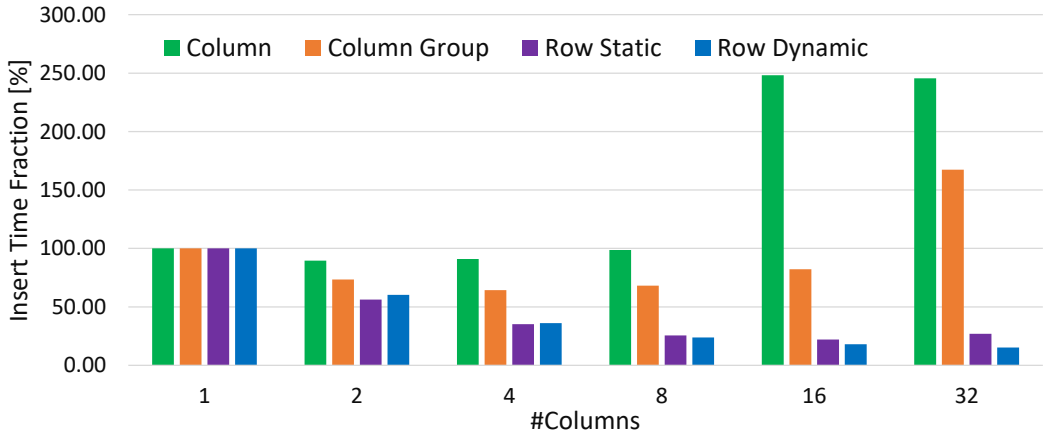


Figure 5.17: Insert time savings of different storage modules depending on the number of grouped attributes (64M elements).

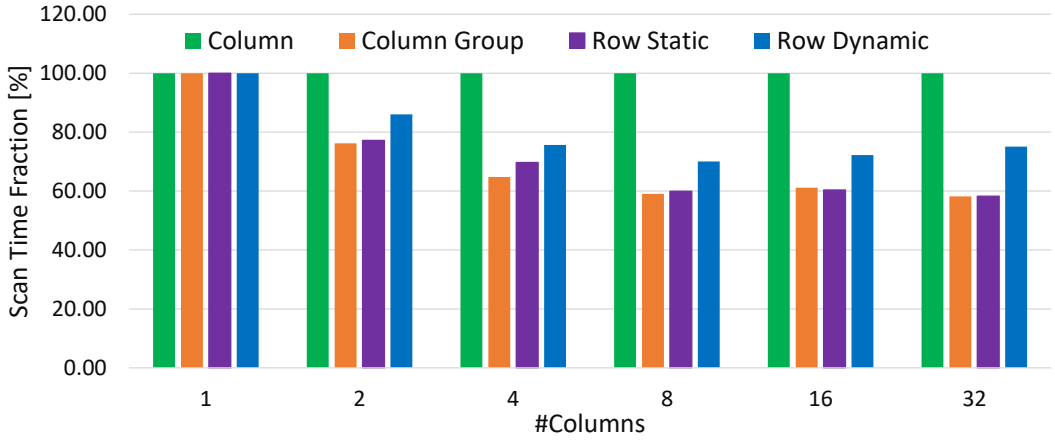


Figure 5.18: Scan time savings of different storage modules depending on the number of grouped attributes (64M elements).

grouping until a certain saturation point is reached where the benefit either stays the same or starts to vanish (e.g., the `Dynamic Row` store). Moreover, we observe that the `Column` store access path gives no grouping advantage, because this feature is not supported by the respective storage module by design.

To summarize, grouping attributes into a single access path is beneficial in most of the cases. However, the benchmarking process also reveals that storage modules can perform worse after a certain number of attribute groupings is reached. Moreover, only storage modules that support more than one attribute can be used to group attributes, which is not the case for our `Column` store implementation. Due

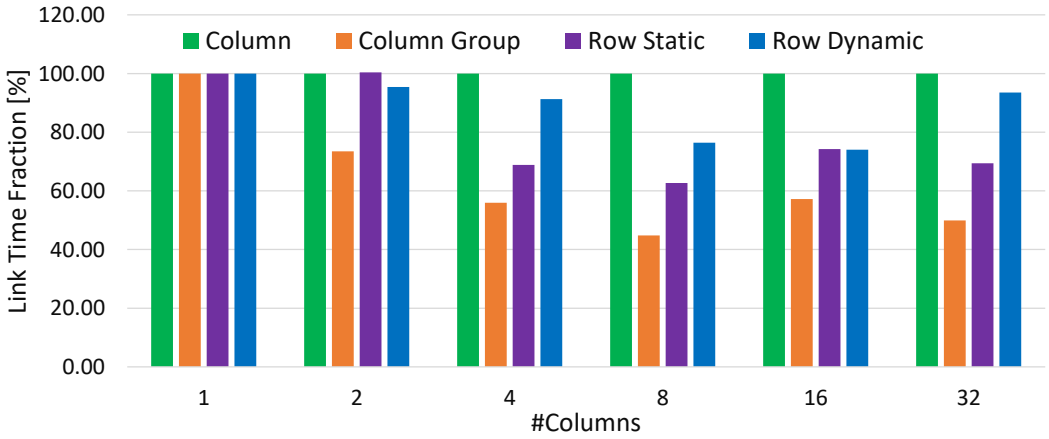


Figure 5.19: Link time savings of different storage modules depending on the number of grouped attributes (64M elements).

to the vast amount of configuration options and possible data as well as request characteristics, we have to limit our benchmarks to the most important supporting points of the huge exploration space to have a lightweight approach for physical storage layout adaptations at runtime.

5.3.2 Storage Layout Controller

The *Storage Layout Controller* is the component of the *Storage ECL* that detects the need for an adaptation on the living partition-level and executes the actual adaption. To make the operation and invocation of the storage layout controller tangible, we depicted an exemplary time line in Figure 5.20. As a starting point, each *1-Storage* instance configures the *cross-world store* to use the bootstrap storage layout, which mainly consists of the *catch-all access path*.

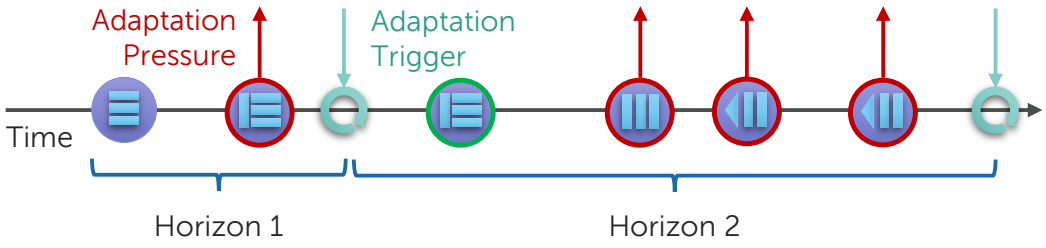


Figure 5.20: Time line showing the storage layout controller calls, the statistics horizon, and actual adaptations.

The storage layout controller is periodically invoked to calculate the *target storage layout*, which happens based on the data and execution statistics gathered since the last adaptation (Horizon 1) as well as the storage module performance statistics.

This invocation is triggered by each request to the 1-Storage that was executed after a certain amount of time was passed since the last invocation. After the target storage layout was computed, the storage layout controller compares it to the current layout of the cross-world store and an *Adaptation Pressure* is reported to the ECL on the higher level, which is calculated according to Formula 5.1. This ECL on the higher level effectively decides on the point of time the actual adaptation happens and triggers the respective storage layout controller to apply the new storage layout. Afterwards, the statistics are reset and the new horizon starts (Horizon 2 in the example). More, the observation horizon is also truncated periodically to evict stale statistics. Since, the actual adaptation is triggered externally, the storage layout controller continues to calculate target storage layouts, which may increases or decreases the reported adaptation pressure. In the following, we will discuss which data and execution statistics are captured, how the target storage layout is computed, and how it is actually adapted.

$$\text{Adaptation Pressure} = t_{\text{processing}} \cdot |\text{storage layout modifications}| \tag{5.1}$$

Data and Execution Statistics

The *μQEP Execution Engine* gathers statistics that are leveraged by the storage layout controller to compute the target storage layout. Moreover, some of the statistics are also used for the *μQEP Compiler*. The respective statistics are either captured per 1-Storage instance or per attribute.

Table 5.5: Per 1-Storage data and execution statistics.

Statistic	Reset	Description
$t_{\text{processing}}$	✓	Time spent for processing 1-Storage requests
#request	✓	Number of requests processed
#records		Number of records inserted
f_{access}	✓	Frequency of attribute access combinations (scans). intersection of *target* and *desired attributes*.

In Table 5.5, we list the statistics that are gathered per 1-Storage instance. The reset column of the table states whether the statistic is reset as soon as a new horizon starts after the actual adaptation or not. For instance, the #records statistic is not reseted, because it is needed to determine the number of records stored in the cross-world store and thus, it is cumulative. In contrast, the other three statistics contain information about the current workload, which is changing over time. For instance, the f_{access} statistic keeps track of frequently queried attribute combinations within the current observation horizon.

In Table 5.6, we enumerate the statistics that are captured per attribute. The #Undefined and Selectivity statistics contain information about the data characteristics and are thus not reset. For instance, the #Undefined statistic is used to

Table 5.6: Per attribute and 1-Storage data and execution statistics.

Statistic	Reset	Description
#Undefined		Number of `UNDEFINED` values seen
#Inserts	✓	Number of `insert` calls on the containing access paths
#Scans	✓	Number of `scan` calls on the containing access paths
#Lookups	✓	Number of highly selective `scan` calls on the containing access paths
#Links	✓	Number of `link` calls on the containing access paths
Selectivity		Selectivity of the attribute. Used for μQEP compilation

determine the sparseness of the attribute, which is done by relating it to the #records statistic. The remaining four statistics once again define the current workload and measure which access primitives were executed on the access path(s) containing the respective attribute. For instance, in the example of Figure 5.9, the #Lookups statistic of the `Age` attribute is incremented and the #Links statistic of the `MVCC`, `Linkage 1`, `Linkage 2`, `Name`, and `Phone` attributes is incremented by the number of partial records that where linked during execution.

Target Storage Layout Computation

To compute the target storage layout, the storage layout controller uses the following three step algorithm. Note that the cross-world store as well as the μQEP Compiler support the redundant storage of attributes in multiple access paths. However, the storage controller is currently limiting redundancy considerations to indexes and we leave this topic open for future work. The optimization goal of the algorithm is to maximize the performance of 1-Storage requests and – as a secondary target – to reduce the memory footprint of a living partition.

Candidate Selection. In the first step, the storage layout controller loops over all attributes and picks all storage modules that are capable of storing the attribute. To do so, the attribute properties are compared to the properties of all access paths (cf., Section 5.3), e.g., to check whether a storage module supports `NULL` values or attributes of a variable length. Afterwards, the candidate list is split into candidates of the value-addressable world and candidates of the other two worlds. Hence, the algorithm so far outputs a *candidate list for indexes* and a *candidate list for primary data structures* per attribute.

Single-Attribute Access Path Selection. To select a suitable access path from the two candidate lists (i.e., indexes and primary storage), the storage layout controller calculates a score for each candidate, which is equal to the estimated execution costs within the current observation horizon. The calculation starts with the primary candidate list, which uses the scoring function in Formula 5.2.

$$\begin{aligned}\text{Score}_{\text{Primary}}(\text{AP}) = \#\text{Links} \cdot t_{\text{Link}}(\text{AP}, \#\text{records}) \\ +\#\text{Inserts} \cdot t_{\text{Insert}}(\text{AP}, \#\text{records}) \\ +\#\text{Scans} \cdot t_{\text{Scan}}(\text{AP}, \#\text{records}) \\ +\#\text{Lookups} \cdot t_{\text{Lookup}}(\text{AP}, \#\text{records})\end{aligned} \tag{5.2}$$

The scoring function calculates the overall costs for the respective access path by using the frequencies of all primitives as well as the performance results of the first stage of the initial benchmarking (cf., Section 5.3.1). Afterwards, the algorithm selects the access path with the *lowest* score as the primary access path. If all scores are zero, because no accesses happened within the observation horizon, the algorithm picks the access path that has the lowest memory consumption considering the sparseness of the attribute. Moreover, such attributes are marked as *unbound*. Following the primary access path selection, the algorithm scores the indexes using the scoring function in Formula 5.3.

$$\begin{aligned}\text{Score}_{\text{Index}}(\text{AP}, \text{AP}_{\text{primary}}) = \\ \#\text{Lookups} \cdot (t_{\text{Lookup}}(\text{AP}_{\text{Primary}}, \#\text{records}) - t_{\text{Lookup}}(\text{AP}, \#\text{records})) \\ -\#\text{Inserts} \cdot t_{\text{Insert}}(\text{AP}, \#\text{records})\end{aligned} \tag{5.3}$$

This scoring function mainly addresses the trade-off between cost savings for `lookups` compared to the primary access path and cost penalties for the additional `inserts`. Afterwards, the algorithm chooses the access path with the *highest* score greater than zero. Hence, one or no index is selected. The result of this step is a *primary access path* and *optionally a secondary access path* for each attribute.

Access Path Grouping. As we demonstrated in Section 5.3.1, the performance of access paths is mostly increased when attributes that are often requested in combination are stored within a single access path. Thus, the algorithm tries to group attributes in a single access path in this final step, which is a non-trivial problem. Hence, we use the greedy heuristics-based approach presented in Algorithm 5.

First, the algorithm fetches all access paths that can contain multiple attributes (line 1) and loops over the most frequent attribute combinations of the f_{access} statistic in a descending order (line 3–11). For each attribute in the combination, the algorithm temporarily elevates the score of the multi column access paths (line 6) using the benchmarking results of the second stage (cf., Section 5.3.1). This elevation function works heuristically and reevaluates Formula 5.2 again while considering the performance savings relative to the #requests the combination is responsible for. Afterwards, those attributes of

Algorithm 5 Grouping step of the target storage layout computation.

```
1: function Step3
2:     APs ← getMultiAttributeAPs()
3:     for each combination ← f_access do
4:         for each attribute ← combination do
5:             if not fixed then
6:                 elevateScores(attribute.candidates, APs, combination.size)
7:             end if
8:         end for
9:         groups ← selectBestMultiAttributeAPs(combination)
10:        markAttributesAsFixed(groups)
11:    end for
12:    groups ← groupUnboundAttributes()
13: end function
```

the combination are grouped that have the same multi attribute access path as the candidate with the lowest score (line 9). All attributes that were grouped are marked as fixed (line 10) and are not considered for additional groupings (line 5–7), which makes the grouping algorithm greedy. Finally, all *unbound* attributes that selected the same multi attribute access path are grouped to reduce the memory consumption (line 12). The result of this final step is a *set of access path configurations* that defines the *target storage layout*.

To summarize, the algorithm employed for the computation of the target storage layout needs to be lightweight to be applicable for online adaptation. As shown in Figure 5.20, the algorithm is called frequently to allow an agile adaptation. Thus, we use a greedy heuristics-based approach that employs a cost model, which is calibrated in the initial benchmarking procedure of the storage modules.

Target Storage Layout Adaptation

To actually apply the target storage layout, a *Living Partition Vitalizer (LPV)* needs to take ownership of the respective living partition whose 1-Storage instance requires the adaptation. During the adaptation process the living partition can not process any other requests. Nevertheless, since living partitions only contain a small fraction of the overall data a relation contains, the adaptation process is fast and queued requests are processed faster on the new physical storage layout. As soon as the executing LPV possesses the respective LP, the storage layout controller compares the target storage layout to the current storage layout and calculates the differences. In the first stage, new access paths are created and populated and in the second stage, outdated access paths are destroyed. Finally, the observation horizon is restarted and the LP is either released or pending requests are immediately processed.

5.3.3 Integration into the ECL Hierarchy

To enable an incremental adaptation of the 1-Storage instances relations are consisting of, it is vital that 1-Storage instances are not adapted all at once. The same holds for 1-Storage instances belonging to different relations. Hence, the adaptation process of the 1-Storage instances, which are not aware of each other, needs to be triggered by an external entity, which mediates the process. Since our ECLs are organized hierarchically, it is a natural decision that this mediation is done by the ECLs on the higher levels of the hierarchy that we introduced in the context of the *Resource Adaptivity-Specific ECL* (cf., Section 4.4). The resource adaptivity-specific ECL employs a single *Global ECL* (cf., Section 4.4.2) and a *Node ECL* (cf., Section 4.4.1) per physical processor. Thus, our goal is to extend the functionality of both ECL types to control the adaptation process of the 1-Storage instances, which run a *Storage ECL*. As a result of this integration, the storage adaptivity-specific ECL covers all ECL types running in the DBMS.

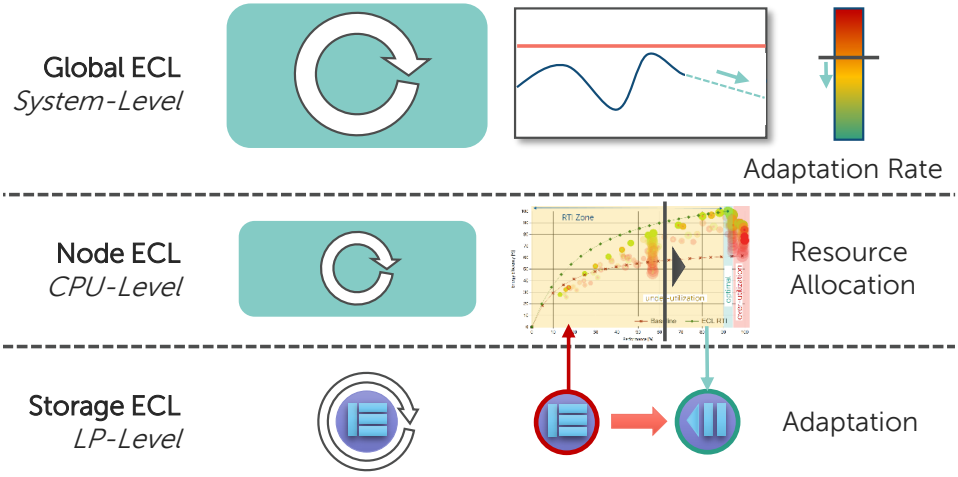


Figure 5.21: Integration of the storage adaptivity-specific ECL into the hierarchy.

In Figure 5.21, we depicted the hierarchy of our different ECL types and focus on how and at which point in time storage layout adaptations are triggered. In the following, we will discuss the responsibilities of the individual ECL types in detail.

Global ECL (System-Level). In the context of resource adaptivity, the global ECL monitors the current average query latency and controls the aggressiveness of the hardware resource allocation, which is controlled by the Node ECL. In the context of storage adaptivity, the global ECL additionally controls the *Adaptation Rate*, which defines the budget for storage adaptations that are allowed to happen within the current execution period. To control the adaptation rate, a similar mechanism is used as for resource adaptivity. Based on the current trend of the average query latency, the adaptation rate is either increased or throttled. If the query latency is increasing or even exceeds the

maximum query latency limit, the adaptation rate is increased proportionally to resolve this severe situation. Otherwise, the adaptation rate is throttled. Nevertheless, to keep the system responsive an upper boundary is defined, which we will determine in our evaluation.

Node ECL (CPU-Level). The primary intention of the Node ECL is to actually configure the hardware resources and to control the number of active LPVs within this context. The storage adaptivity-specific ECL extends this functionality by registering the *Adaptation Pressure* reported by the storage ECLs. Within each execution cycle, the Node ECL orders the requests in a descending order and tries to obtain a portion of the available global adaptation budget (adaptation rate). Based on the retrieved budget, an equal number of adaptation requests is approved and triggered. Moreover, the Node ECL considers the amount of triggered adaptations for its hardware resource allocation.

Storage ECL (LP-Level). As we already discussed throughout this section, the storage ECL mainly consists of the *Storage Layout Controller*, which checks the current storage layout of the cross-world store for necessary adaptations in a query-driven way and reports a certain adaptation pressure to the Node ECL. As soon as the Node ECL approves the adaptation request, the actual adaptation is executed.

To summarize, 1-Storage instances require a higher level entity to coordinate the storage adaptation process to avoid a high amount of concurrent adaptations, which hurt the responsiveness of the database system. Hence, the storage adaptivity-specific ECL leverages and extends the functionality of the ECLs implemented by our resource adaptivity approach. The most important knob is the global adaptation rate, which controls the amount of storage adaptations that happen simultaneously and thus, represents a trade-off between query responsiveness and adaptation efforts that positively affect the future query execution performance and energy efficiency of the DBMS.

5.4 Evaluation

In this section, we evaluate several aspects of our *Storage Adaptivity* approach using an implementation of *1-Storage* and the *Storage Adaptivity-Specific ECL* in *ERIS*. Since *1-Storage* needs to build a μQEP for each request, we first investigate to which degree those additional costs can be compensated by the μQEP cache. Afterwards, we demonstrate the adaptation behavior of our storage adaptivity approach using a basic scenario and focus on the *Adaptation Rate*, which defines how many *Living Partitions* are adapted within the current interval of the *Global ECL*. The next series of experiments addresses the combination of *Resource Adaptivity* and *Storage Adaptivity* in the context of energy consumption. We will demonstrate that both *Adaptivity Facilities* work hand in hand and each of them contributes to the *Energy*

Awareness of the database system. Finally, we will use a workload mix to investigate the adaptation behavior of our approach in the context of a shifting workload. We conducted all experiments on the SGI UV 3000 system, except for the energy-related experiments that were executed on the 2-socket Haswell-EP system (cf., Table 3.1).

5.4.1 Micro QEP Cache

Each request to a 1-Storage instance requires the compilation of a μQEP, because the physical storage layout can change during the execution of a query. This compilation induces a significant overhead, especially for requests that have a short execution time (e.g., a highly selective lookup). Hence, we reconsider our microbenchmark of Figure 3.29 in Section 3.4.6, which queries and indexed key-value store with a scale factor of 1000. In this microbenchmark, ERIS needs to process up to 200 million requests per second on its 1-Storage instances to execute an equal number of queries on the key-value store. Moreover, the experiment investigated the impact of message coalescing and the locality of requests using a local (all requests of a task to the local processor), shifted (all requests of a task to the same foreign processor), or random (all requests of a task to a random processor) pattern.

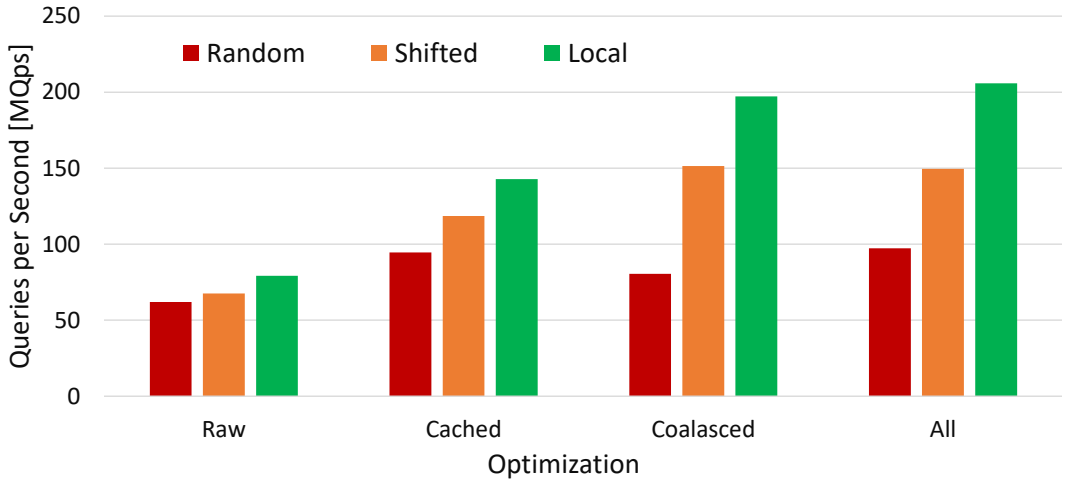


Figure 5.22: Impact of the micro QEP cache and coalesced message processing.

While the original experiment already used an activated μQEP cache, we will now show the measurements without an active μQEP cache. In Figure 5.22, we visualized the peak throughputs (all 768 cores activated) for the different optimization techniques (i.e., μQEP cache and message coalescing) and the different message locality patterns. Without any optimization technique (Raw), the performance results are the lowest and differ only slightly among the different locality patterns. Moreover, we measured that **50 %** of the request processing time accounts to the μQEP compilation. Thus, an ideal μQEP cache should double the performance numbers for

the individual locality patterns. As the measurements for an activated μQEP cache show, the cache can almost fully eliminate the μQEP compilation overhead.

The benefit of the message coalescing in the context of the μQEP compilation is that only a single μQEP needs to be compiled for a batch of messages. Hence, we achieve the best results of this optimization technique for the local and shifted locality pattern, because all requests of a task are send to the same processor, which leads to big message respectively request batches. For that reason, we only get a little advantage by additionally activating the μQEP cache. However, for the random locality pattern, message coalescing is almost impossible and thus, we get most of the performance gains from the μQEP cache here.

5.4.2 Adaptation Rate

To enable an incremental adaptation of the physical storage layout, a 1-Storage instance can not trigger its adaptation on its own. For instance, if all 1-Storage instances of a relation simultaneously decide to do an adaptation, the entire query processing grinds to a halt. Hence, the global ECL defines a certain budget of LPs that can be adapted within the current execution interval and the Node ECLs distribute this budget between their LPs that registered for a pending adaptation. This budget is called the *Adaptation Rate*, because it defines the number of LPs that can be adapted relative to the number of active LPVs, which can be deactivated due to resource adaptivity. In the following, we stay with the basic but expressive key-value store example (the same pattern occurs for joins) and present time slices of the query execution where the storage layout of the 1-Storage instances is adapted from the default layout (catch-all access path only) to the best fitting storage layout. This storage layout consists of a column store access path for each attribute (i.e., `key`, `value`, and `MVCC`) and an additional secondary RB-Tree access path on the key attribute. In the experiments, we use a scale factor of 1000 (100 M keys; 3.2 GB raw data) and 5000 (500 M keys; 16 GB raw data).

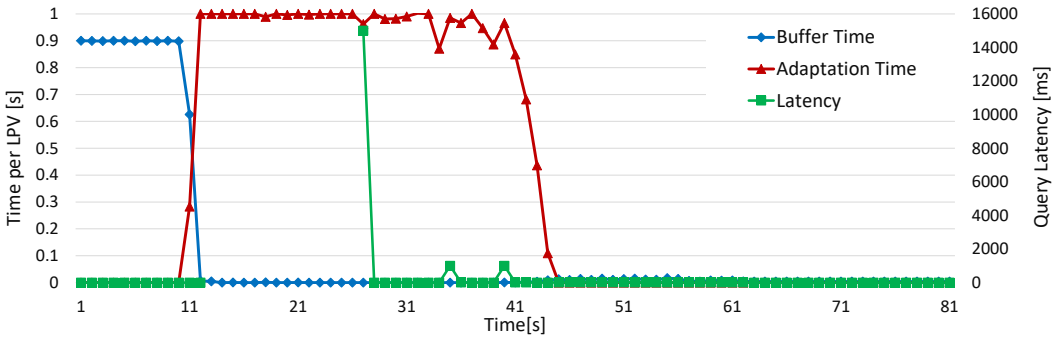


Figure 5.23: Adaptation behavior for an adaptation rate of 100 % (SF 1000).

In Figure 5.23, we fixed the adaptation rate to 100% for an execution interval of the global ECL of 1 second, which amounts to 768 LPs per second. Since the

key-value store is also partitioned into 768 LPs, the entire relation is adapted at once and thus, this scenario is our baseline, because it mostly equals to a full table lock and rebuild. Before the adaptation is started, the average query latency is at about 3ms and the system is utilized to 90%. As soon as the storage adaptation process is triggered for all LPs at once, the system becomes totally unresponsive for 16 seconds and afterwards responds at a high latency, because some of the LPs already finished their adaptation and continued to process queued requests. As the chart shows, the buffer time per LPV, which is the time spent for processing LP requests, starts at 0.9 s and drops to 0 s during the initial phase of the adaptation and the adaptation time fully dominates the work of an LPV. After those first 16 s of the adaptation, the latency is varying between 5 μs and 1 s and stabilizes at 5 μs about 40 s later, because those LPs that finished their adaptation latest are still processing their pending requests. Hence, an adaptation rate of 100 % is not suitable for an online storage adaptation, because it massively affects the query execution and leads to unpredictable query latencies. Nevertheless, we were still able to reduce the average query latency from 3 ms to 5 μs and decreased the system load from 90 % to <1 %.

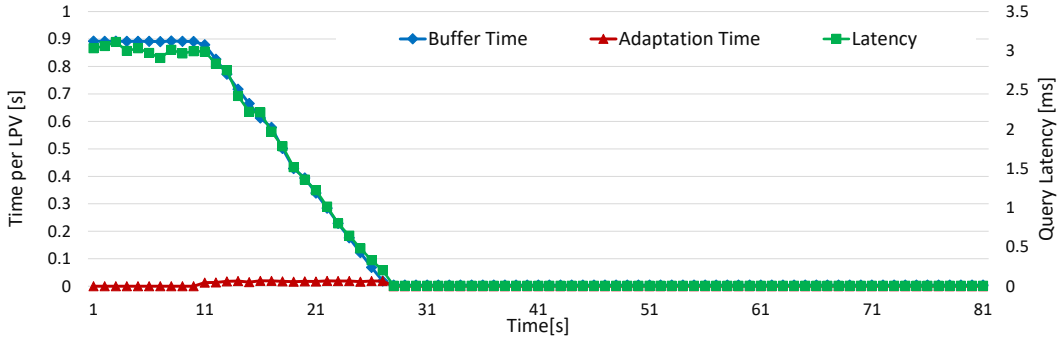


Figure 5.24: Adaptation behavior for an adaptation rate of 6.25 % (SF 1000).

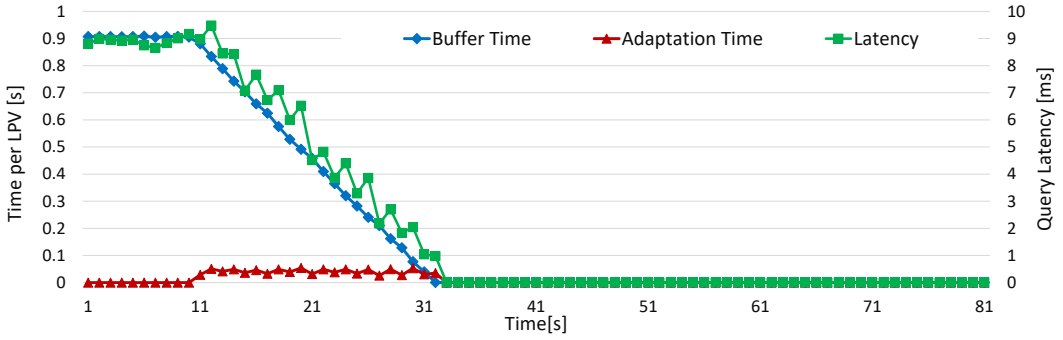


Figure 5.25: Adaptation behavior for an adaptation rate of 6.25 % (SF 5000).

In Figure 5.24 (SF 1000) and 5.25 (SF 5000), we pinned the adaptation rate to 6.25 %, which equals to 48 LPs per second. Both scale factors employ 768 LPs and

hence, LPs in the SF 5000 scenario (20.8 MB raw data per LP) are five times as big as in the SF 1000 scenario (4.2 MB raw data per LP). As the measurements show, ERIS stays responsive during the adaptation process and the query latency smoothly decreases from 3 ms to 5μs. In the SF 5000 scenario, we observe small latency peaks that occur periodically during the adaptation, because of the bigger LP sizes. Nevertheless, in this scenario, we were also able to reduce the query latency from 9 ms to 5μs without a significant initial overhead. Moreover, we noticed that the actual adaptation takes only 16 s (SF 1000) respectively 20 s (SF 5000) compared to 33 s in the 100 % adaptation rate case.

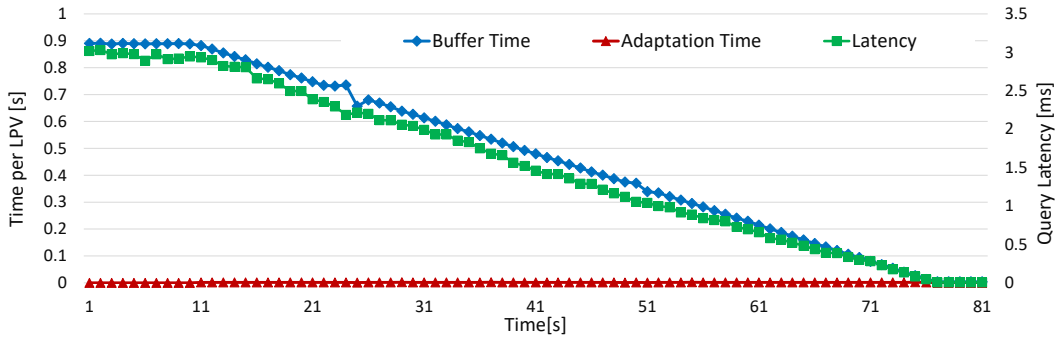


Figure 5.26: Adaptation behavior for an adaptation rate of 1.6 % (SF 1000).

In Figure 5.26 (SF 1000), we fixed the adaptation rate to 1.6 %, which equals to 12 LPs per second. The only difference we observe compared to the previous scenario is that the adaptation takes longer (64 s) and that the adaptation time of the LPVs is almost not noticeable.

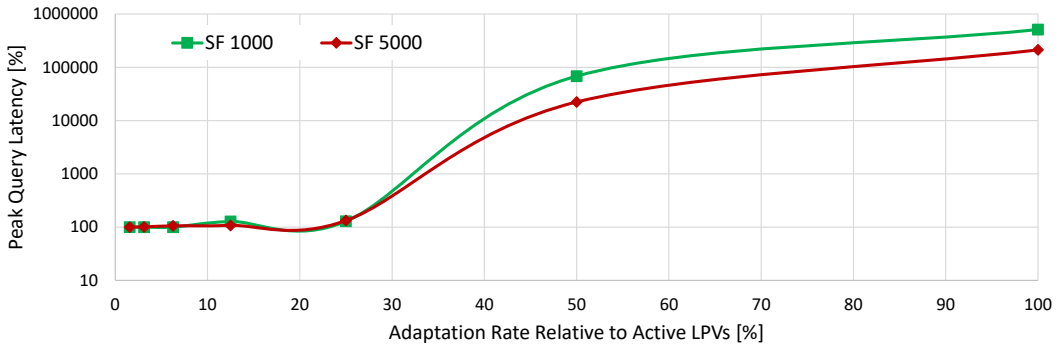


Figure 5.27: Relative peak latency depending on the adaptation rate.

As the previous experiments showed, the adaptation rate mainly affects the *peak query latency* and the *duration* of the adaptation process. Hence, we compared those two key factors for both scale factors (i.e., 1000 and 5000) depending on the adaptation rate in Figure 5.27 (peak query latency relative to the pre-adaptation latency) and 5.28 (duration). As the measurements show, the initial latency overhead

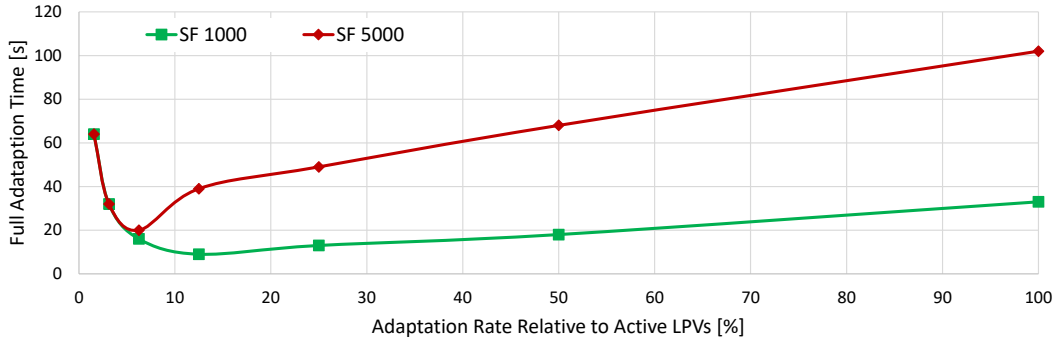


Figure 5.28: Adaptation duration depending on the adaptation rate.

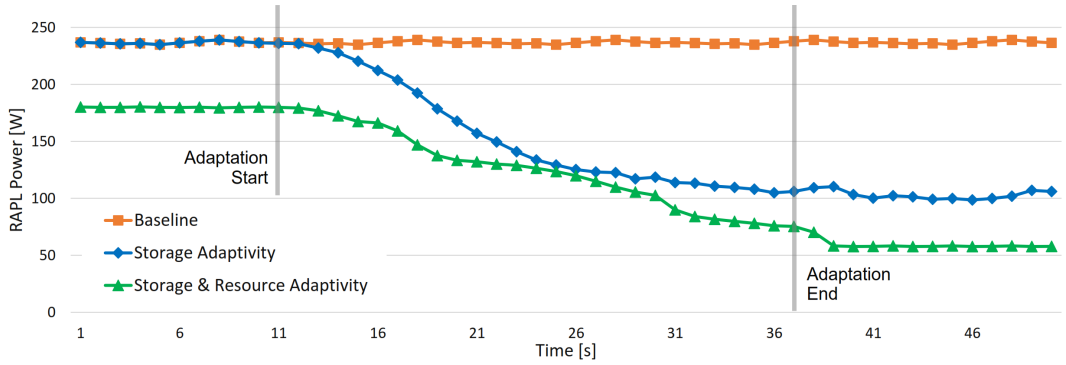
is almost not there until an adaptation rate of 6.25 % is passed. For an adaptation rate of 25 % we already observe a latency overhead of about 30 % for both scale factors. Beyond this adaptation rate, the latency overhead becomes unjustifiable high, because tasks and requests queue up and hardware threads pollute the caches, which also causes more efforts of the cache coherency protocol on such a large-scale NUMA system. We observe a similar break-even point for the adaptation duration. However, this point slightly depends on the LP size, because larger LPs may not finish within the execution interval of the global ECL and the adaptation budget calculation also considers ongoing adaptations.

To summarize, the *Adaptation Rate* has a significant impact on the storage adaptation process. As our evaluation showed, it mainly affects the *peak query latency* and the *duration* of the actual adaptation. Based on our measurements, we identified a maximum adaptation rate of 6.26 %, which is equal to a sixteenth of the active LPVs, as feasible.

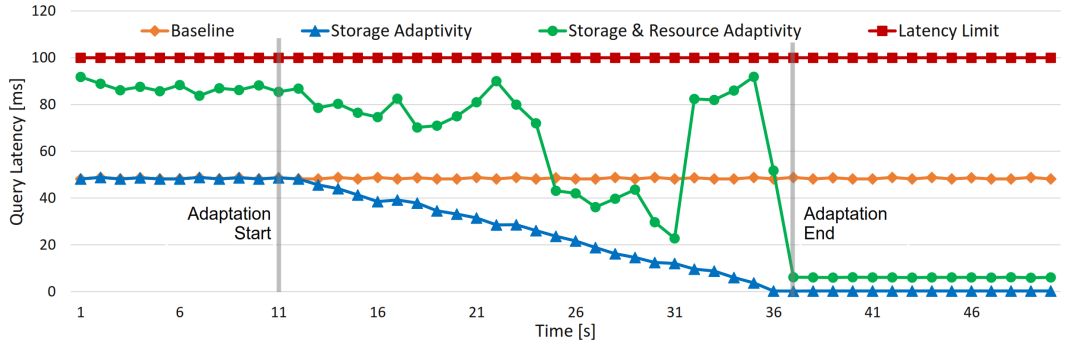
5.4.3 Energy Savings

As the previous experiments revealed, our *Storage Adaptivity* approach is able to dramatically decrease the query latency as well as the system utilization. Since the main objective of this thesis is to reduce the energy footprint of database systems, we will now quantify the energy savings of our *Storage Adaptivity* approach in isolation as well as in combination with our *Resource Adaptivity* approach. For this setup, we use the 2-socket Haswell-EP system (cf., Table 3.1) and a scale factor of 200 (20 M keys; 640 MB raw data size; 48 LPs) for the key-value store.

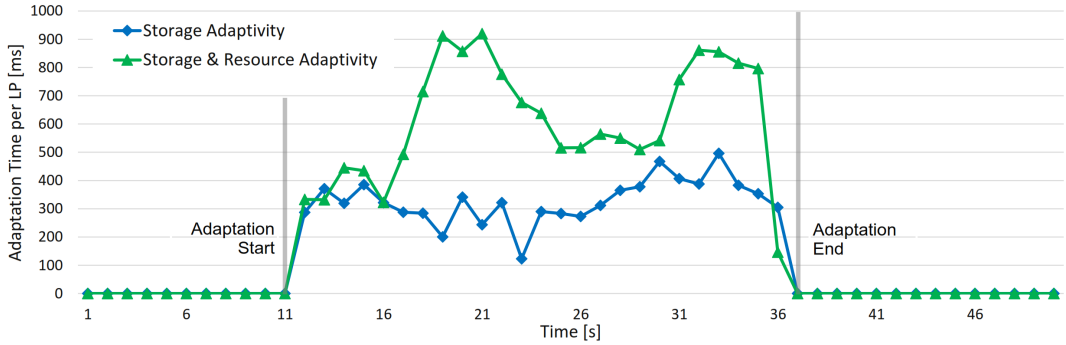
In Figure 5.29, we visualized the results for the *baseline* (all adaptivity facilities turned off), *Storage Adaptivity*, as well as *Storage and Resource Adaptivity*. In the storage adaptivity-only scenario, the adaptation rate is fixed to 2 LPs per second and in the storage and resource adaptivity scenario, the adaptation rate is controlled by the *Global ECL* using a maximum of 3 LPs per second based on the results of the



(a) Power consumption over time.



(b) Query Latency over time.



(c) Time spent for adaptation per LP over time.

Figure 5.29: Power draw, query latency, and adaptation efforts using different adaptivity facilities.

respective evaluation. Figure 5.29(a) demonstrates that the baseline has a constant power draw of 240 W (RAPL measurements). In contrast, storage adaptivity starts with the same power consumption, but is able to reduce the power draw to 100 W after the adaptation finished. Additionally enabling resource adaptivity already

decreases the power consumption before the storage adaptation happens and ends up with a power draw of only 60 W.

Our resource adaptivity approach accomplishes this lower power draw by trading power for an increased query latency as shown by Figure 5.29(b). As the chart shows, the query latency of our baseline remains constant while the storage adaptivity-only scenario continuously decreases the latency solely by incrementally adapting the physical storage layout. If we additionally activate the resource adaptivity, we observe a varying latency that always stays below the preset latency limit of 100 ms, but is always higher compared to the storage adaptivity-only case. Moreover, we observe sudden peaks within this latency, which are caused by the switch to lower operating frequencies of the processor or changes in the adaptation rate. We observe the same effect in the time spent for adaptations by an LP respectively LPV (relative to the current adaptation rate) shown in Figure 5.29(c). The first peak (around 20 s) is caused by an increase of the adaptation rate, because the query latency was too close to the maximum, which caused the global ECL to take this action. The second peak is a result of a switch to lower processor frequencies, which causes the adaptation times to increase.

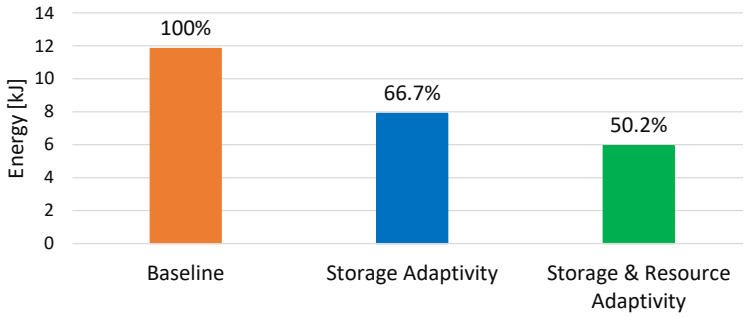


Figure 5.30: Total energy savings using different adaptivity facilities.

In Figure 5.30, we give the overall energy consumption numbers for the three setups as well as the relative energy consumption compared to the baseline. As shown, by only enabling storage adaptivity we already save 33.3 % energy and by additionally activating resource adaptivity we measured total energy savings of about 50 %. Since current hardware is still far away from being energy proportional, our resource adaptivity is limited in its opportunities for such low system utilizations as they happen after the storage adaptation.

To summarize, we demonstrated that our two *Adaptivity Facilities* contribute to increase the energy awareness of a database system by either doing software-centric adaptations (*Storage Adaptivity*) or hardware-centric adaptations (*Resource Adaptivity*). Moreover, we showed that both unleash their full potential when being

employed in combination. For our specific workload, we were able to save about 50 % energy while additionally improving the query latency by orders of magnitude.

5.4.4 Workload Mix

In our final experiment, we evaluate our *Storage Adaptivity* approach for a workload that is changing over time. As foundation, we use the `Orders` table of the TPC-H benchmark [132], which consists of 9 attributes as shown in Table 5.7, and three extreme workloads: (1) WL_{Insert} is a transactional load workload that inserts records into the table, (2) WL_{Lookup} is a transactional workload that looks up orders by their `ORDERKEY` requesting all attributes, and (3) WL_{Scan} is an analytical workload that aggregates the `TOTALPRICE` column. Initially, the `Orders` table is empty and we execute the workloads in the following sequence: WL_{Insert}, WL_{Lookup}, WL_{Insert}, and WL_{Scan}. The adaptation rate is fixed to 6.25 % and the experiment was conducted on the SGI UV 3000 (cf., Table 3.1).

Table 5.7: Orders table schema and physical storage layout for the different workloads. *Row Store Static(RSS)*, *Column Store (CS)*, *RB-Tree (RB)*.

Attribute	Data Type	WL_{Insert}	WL_{Lookup}	WL_{Scan}
`ORDERKEY`	BIGUINT	RSS_1	RSS_1, RB_1	RSS_1
`CUSTKEY`	BIGUINT	RSS_1	RSS_1	RSS_1
`ORDERSTATUS`	CHAR(1)	RSS_1	RSS_1	RSS_1
`TOTALPRICE`	DECIMAL	RSS_1	RSS_1	CS_1
`ORDERDATE`	DATE	RSS_1	RSS_1	RSS_1
`ORDERPRIORITY`	CHAR(15)	RSS_1	RSS_1	RSS_1
`CLERK`	CHAR(15)	RSS_1	RSS_1	RSS_1
`SHIPPRIORITY`	INT	RSS_1	RSS_1	RSS_1
`COMMENT`	VARCHAR(79)	RSS_1	RSS_1	RSS_1

In Figure 5.31, we visualized the adaptation behavior of *ERIS*, which implements our storage adaptivity approach. The chart shows the measurements for the *query latency*, the *buffer time* (time consumed for processing LP requests), and the *adaptation time*. Note that latency and buffer times are given relative to the baseline, which stays with the default *catch-all access path* storage layout.

As soon as the WL_{Insert} workload is executed for the first time, all *1-Storage* instances calculate a new *target storage layout*, which moves all attributes from the catch-all access path (dynamic row store) to a new static row store access path as shown by Table 5.7. This decision is made, in the grouping stage of our algorithm (cf., Section 5.3.2), because the `COMMENT` attribute is of a variable size and thus, a column group access path can not be considered for the attribute grouping. Since this adaptation affects all attributes including the MVCC attribute, a full 1-Storage reorganization is necessary, which comes at high costs, because the dynamic row store has a bad scan performance and the actual adaptation process also uses the *Data*

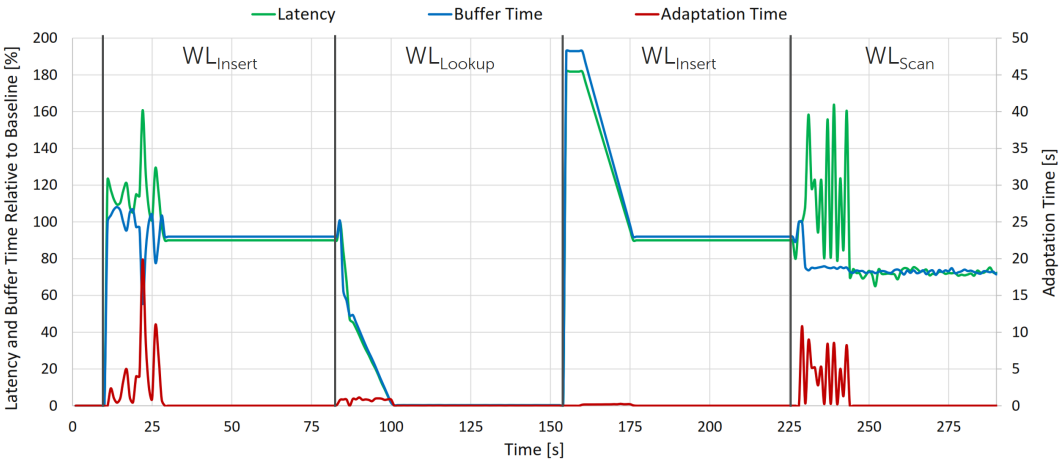


Figure 5.31: Adaptation behavior over time for the workload mix.

Gateway as indirection layer. After a short adaptation period, the query latency as well as the buffer time are reduced by 10 %. Moreover, the adaptation reduced the memory footprint of the LP, because the static row store version consumes less memory compared to its dynamic pendant.

Afterwards, the switch to the WL_{Lookup} workload occurs. The 1-Storage instances quickly detect that the `ORDERKEY` is often queried in a highly selective fashion and due to the currently small partition sizes, the lookup savings dominate the insert costs, which are still present in the *observation horizon*. Hence, the 1-Storage instances additionally build an index on this attribute as the physical storage layout in Table 5.7 shows. The chart demonstrates that the actual adaptation costs are very low and the latency and buffer time advantages are tremendous. However, as we switch back to the WL_{Insert} workload, this index increases the insert costs and the trade-off between insert costs and lookup savings still favors the presence of the index. Hence, the physical storage layout remains the same until the periodic truncation of the observation horizon happens and the LPs start to destroy the index, which is cheap in terms of adaptation costs. Due to the adaptation, latency and buffer time smoothly decrease to the level of the first WL_{Insert} run.

Finally, we switch to the analytical WL_{Scan} workload and the 1-Storage instances quickly detect a scan access pattern on the `TOTALPRICE` attribute and decide to move this attribute to a column store access path as shown in Table 5.7. Hence, the adaptation process builds this access path, but additionally needs to rebuild the static row store without the `TOTALPRICE` access path as well as its *Linkage* column. In this scenario, we observe that the buffer time quickly drops to its final level and that the latency stays higher until the adaptation finishes. This behavior occurs, because the scan is broad-casted to all LPs of the relation and the performance depends on the slowest LP. Hence, the query latency decreases not before all LPs finished the adaptation process and in the meanwhile, the latency overhead mirrors the adaptation costs. The same effect happens for lookups that need to be broad-

casted because of an incompatible partitioning scheme. As result of the adaptation, latency and buffer time improve by more than 20 % compared to the baseline. Note that without the indirection induced by the *Data Gateway*, this benefit would be even higher. The final raw data size of the table was about 2.5 GB and for larger tables the adaptation process takes a proportional amount of time assuming that the adaptation rate remains the same. Nevertheless, as the table gets bigger the LP sizes get bigger too and at a certain point in time, LPs must be split to keep the storage layout adaptation incremental and its invasiveness low. This issue is a matter of *Data Placement Adaptivity*, which is subject of future work.

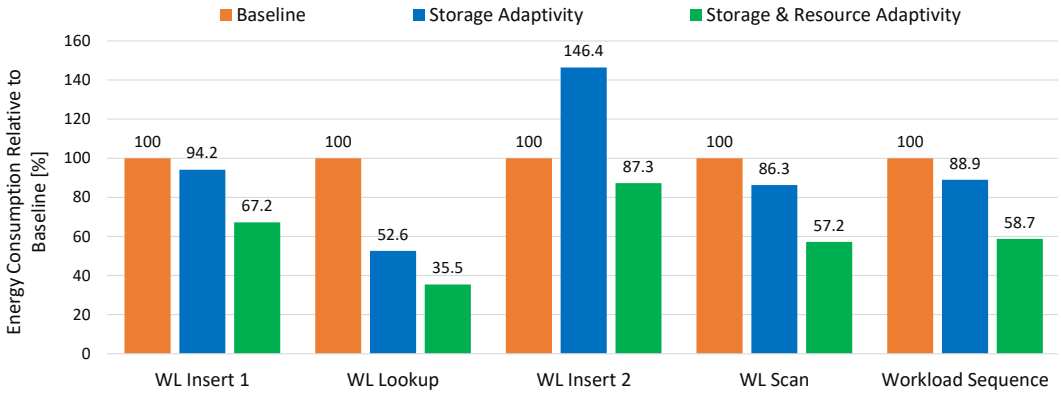


Figure 5.32: Energy savings respectively overhead of the adaptivity facilities for the workload mix.

We executed the same workload sequence on the 2-socket Haswell-EP system (cf., Table 3.1) to measure the energy consumption using our adaptivity facilities. The workload was scaled down to the smaller machine and the system utilization for the baseline was about 90 %. In Figure 5.32, we visualized the energy measurements relative to the baseline. For the first execution of the $\text{WL}_{\text{Insert}}$ workload, storage adaptivity saves only a small amount of energy, because of the low optimization potential. However, in combination with resource adaptivity, the overall energy savings amount to more than 30 %. In contrast, for the second run of the same $\text{WL}_{\text{Insert}}$ workload, we measure an energy *overhead* of about 46 %, because of the additional costs for maintaining the index that is present for a certain amount of time. Those additional costs can not be compensated within the short execution phase of the workload. Nevertheless, the additional hardware adaptations made by our resource adaptivity approach result in more than 10 % energy savings. We experienced the highest energy savings during the execution of the $\text{WL}_{\text{Lookup}}$ workload, because of the huge optimization potential of the physical storage layout. Our storage adaptivity approach is able to quickly detect and execute the adaptations at low costs. Moreover, the additional activation of our resource adaptivity results in total energy savings of about 65 %. Finally, during the execution of the WL_{Scan} workload, storage adaptivity is able to save about 14 % energy. Because our resource adaptivity

approach detected a changing workload type, which is now memory-bound, it adapts its *Energy Profile* (cf., Section 4.4.1) and is able to additionally save a high fraction of energy. As the relative numbers for the full workload sequence show, storage adaptivity in isolation is only able to save roughly 11 % of the energy due to the high overhead for the second execution of the $\mathrm{WL}_{\mathrm{Insert}}$ workload. This is a result of the lack of information about the future workload and the delayed adaptation that happens consequently. However, the combination of both adaptivity facilities is able to save more than 40 % of energy. This experiment emphasizes the potential of our current approach as well as its limitations, which require additional research efforts to be overcome.

To Summarize, our evaluation demonstrated that our *Storage Adaptivity* approach is able to quickly adapt to changing workloads during DBMS operation, which reduces the buffer time and thus, decreases the energy footprint of the system. Our investigations also emphasized that our approach needs additional augmenting techniques to unfold its full potential. Those techniques are just-in-time compilation to eliminate the indirection overhead, forecasting to consider the trade-off between adaptation costs and adaptation benefit, redundancy, as well as *Data Placement Adaptivity* to manage the LP sizes.

5.5 Summary and Conclusions

Modern application scenarios require data management systems to cope with a vast variety of datasets and query types that are not known beforehand and change over time. To still provide a superior performance and energy efficiency in all of the potential scenarios, the database system needs to adapt its physical storage layout to the current workload, because the data organization has a significant impact on the query execution performance and there exists no one-size-fits-all physical storage layout. Current solutions for the storage adaptation problem are very limited, because they are either designed as offline approaches or address only a small subset of the available storage layout tuning knobs.

In this chapter, we presented *Storage Adaptivity* as a holistic software-centric approach for increasing the performance and energy efficiency of a database system in the presence of varying workloads and data characteristics. Our approach consists of two main components. The first component is *1-Storage*, which is a storage manager that is able to organize the data in a multitude of physical layouts combining the advantages of row-wise and columnar data organizations in combination with adaptive indexing. Furthermore, 1-Storage uses the concept of extensible storage modules to provide support for any kind of access path implementation. Since 1-Storage is designed to operate within the *Living Partitions* architecture, each living partition of a relation is additionally able to apply a different physical storage layout. The *Storage Adaptivity-Specififc Energy-Control Loop* leverages this implicit partitioning of the architecture for enabling a fine-grained incremental adaptation of the physical storage layout in case of a changing workload. Moreover, this component integrates well with our *Resource Adaptivity-Specific ECL*, which results in a sophisticated ECL hierarchy that is able to control the adaptation of hardware as well as software at runtime while trying to stay within a user-defined query latency limit.

Our evaluation demonstrated that all ECLs work hand in hand and we observed energy savings of about 65 % while additionally improving the query latency by orders of magnitude. Nevertheless, we also revealed the limitations of the current state of our approach and suggested just-in-time compilation, forecasting, redundancy, and *Data Placement Adaptivity* as augmenting techniques to overcome these limitations.

6 Summary and Conclusions

Recent studies revealed that the energy consumption of server hardware already became a critical problem, especially in data centers. Since data management systems are an application class that amounts to a high portion of the overall deployments, they are responsible for a high share of the energy draw. Another trend is the ongoing move from disk-based to in-memory database systems, which run on hardware that exhibits more and more non-uniform memory access (NUMA) related effects. In this thesis, we investigated how the energy consumption of such in-memory database systems that run on mid and large scale-up NUMA hardware platforms can be reduced.

In the first place, we discussed the nature of energy in the context of data management systems and derived the term energy awareness as the ability of a DBMS to actively optimize its energy efficiency as well as energy proportionality. We came up with our core concept of *Energy Awareness by Adaptivity*, which aims at active software-driven adaptations at runtime, especially in the presence of changing workloads and data characteristics as it is becoming increasingly common in today's applications. To actually implement this concept, we defined a rich set of requirements that need to be fulfilled to build an energy-aware database system. Those requirements either originate from the general scalability prerequisite or from the ability to enable fine-grained adaptations at runtime.

Our exploration of existing database architectures concluded that none of the known architectures fulfills our requirements for an energy-aware DBMS. Nevertheless, we decided to use the *data-oriented architecture* as a starting point, because of its scalability advantages. We prototypically optimized this architecture for large scale-up in-memory database systems and achieved excellent scalability results as well as absolute performance numbers that clearly outperform the traditional transaction-oriented architecture. To especially enable our *Adaptivity Facilities* on this architecture, we proposed the *Living Partitions* architecture that treats *Living Partitions* as autonomous self-adapting objects and presented the database system *ERIS* that is based on this novel architecture. Our evaluation showed superior scalability of ERIS for transnational workloads on a large scale-up NUMA system.

Using ERIS and the living partitions architecture as a solid foundation, we investigated two *Adaptivity Facilities* that implement our core concept. The first implementation is the hardware-centric *Resource Adaptivity*, which actively adapts the hardware configuration by controlling the rich set of available energy-control knobs of current processors. Our resource adaptivity approach implements the *Resource Adaptivity-Specific Energy-Control Loop (ECL)*, which consists of a system-level *Global ECL* and a CPU-level *Node ECL*. While the global ECL keeps track of

the current average query latency, the Node ECL maintains an adaptive *Energy Profile* to manage the hardware configurations. In our evaluation, we measured energy savings ranging from 20% to 40% for a real-world load profile.

The second *Adaptivity Facility* we investigated was *Storage Adaptivity*, which is a software-centric approach for adapting the physical storage layout at runtime. Our approach uses the extensible *1-Storage* storage manager that is capable of organizing its data in a wide variety of physical representations covering columnar and row-wise data organizations as well as adaptive indexing. To actually adapt the physical storage layout, we presented the *Storage Adaptivity-Specific Energy-Control Loop*, which leverages the implicit partitioning of our living partitions architecture to incrementally adapt the storage layout at runtime. We described how to integrate the different ECLs with each other and ended up with a sophisticated ECL hierarchy that is doing hardware and software-centric adaptations at runtime, while trying to stay within a user-defined query latency limit. Our evaluation showed that all ECLs work hand in hand and we achieved superior energy savings and query latency improvements for various workload mixtures.

Future Work

In our opinion, highly adaptive database systems are the only way to cope with the vast amount of application domains database systems are being exposed today. The scalable and adaptivity-enabling *Living Partitions* architecture as well as our *Adaptivity Facilities* are a first milestone towards such a highly adaptive DBMS, which opens up new horizons for further research. In the following, we discuss further promising directions for research that we are currently thinking of.

Resource Adaptivity. A major weakness of our *Resource Adaptivity* approach is its limitation to short-running queries, because the reactive design of our global ECL requires a fast feedback. In contrast, the Node ECL works even with long-running queries, but receives no feedback about the current query latency and thus, the decisions made are maybe far away from the optimum. To cope with this issue, we need to be able to track the progress of such queries to estimate their latency. Another point addresses the generation and maintenance of our *Energy Profiles*. Due to the huge exploration space of possible hardware configurations, which is getting bigger and bigger as the number of energy-control knobs grows, we need more sophisticated techniques for maintaining such profiles.

Storage Adaptivity. A problem of our *Storage Adaptivity* approach is the indirection layer, which adds a significant overhead to the query processing as our evaluation showed. To cope with this issue, we think of *Code Adaptivity* as an additional *Adaptivity Facility* that employs just-in-time compilation to specialize the DBMS code at runtime. Moreover, the same technique is applicable for speeding up the actual adaptation process by specializing the transformation

between storage formats. However, such an approach adds additional compilation costs that need to be considered. To actually consider the one-time costs for compilation and adaptation, we need the ability to look into the future to decide whether the costs will amortize or not. Simply waiting for a long time, before actually paying those one-time costs is bad solution, because a lot of energy is potentially wasted in the meantime. Forecasting techniques are a good starting point to get a rough hint of how the workload will behave in the future. Another issue that we left open for future work is the redundancy of data beyond indexes when computing a new target storage layout.

Data Placement Adaptivity. This *Adaptivity Facilitiy* was already proposed and prototypically evaluated by us, but is not further considered in this thesis. To enable data placement adaptivity as we described it for our *Living Partitions* architecture, we need the appropriate tooling and a lightweight decision making to keep the entire process agile. Moreover, we need to find efficient ways for splitting a *1-Storage* instance in case of a *Living Partition* split. Similar to *Storage Adaptivity*, redundancy is also a desirable goal for hardening the DBMS against workloads and datasets that are hard to partition.

Self-Learning Adaptation Mechanisms. When we think of all of those *Adaptivity Facilities* that need to work hand in hand, we will run into problems, because of the vast amount of dependencies between them and the side effects that can occur and we will reach the point where developers are not able to understand them anymore. Hence, we have to think of self-learning decision models such as deep learning or other kinds of artificial intelligences that do the actual adaptation decisions based on the past experience.

Bibliography

[1] Inauguration Day on Twitter. [Online] `https://blog.twitter.com/2009/inauguration-day-on-twitter`.

[2] likwid. `http://code.google.com/p/likwid/`.

[3] Linux Perf Wiki. [Online] `https://perf.wiki.kernel.org`.

[4] SFB 912: Highly Adaptive Energy-Efficient Computing. [Online] `https://tu-dresden.de/ing/forschung/sfb912`.

[5] VAMPIRTRACE. `http://www.tu-dresden.de/zih/vampirtrace`.

[6] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom. The Beckman report on database research. *Commun. ACM*, 59(2):92–99, 2016.

[7] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682, 2006.

[8] S. Acharya, P. Carlin, C. A. Galindo-Legaria, K. Kozielczyk, P. Terlecki, and P. Zabback. Relational support for flexible schema scenarios. *PVLDB*, 1(2):1289–1300, 2008.

[9] Advanced Micro Devices. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors. 2012.

[10] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[11] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114, 2014.

[12] O. Arnold et al. An Application-Specific Instruction Set for Accelerating Set-Oriented Database Primitives. In *SIGMOD*, 2014.

[13] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598, 2016.

[14] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[15] R. Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972.

[16] B. M. Beckmann and D. A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *MICRO*, pages 319–330, 2004.

[17] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 58, 2006.

[18] M. Bhandaru et al. Dynamically controlling interconnect frequency in a processor. [Online]: http://www.google.com/patents/ WO2013137862A1?cl=en.

[19] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[20] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, 2011.

[21] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB J.*, 8(2):101–119, 1999.

[22] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[23] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65, 1999.

[24] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[25] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[26] P. J. Brockwell and R. A. Davis. *Introduction to Time Series and Forecasting*. Springer, 3 edition, 2016.

[27] R. E. Brown, E. R. Masanet, B. Nordman, W. F. Tschudi, A. Shehabi, J. Stanley, J. G. Koomey, D. A. Sartor, and P. T. Chan. Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431. 06/2008 2008.

[28] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *IJHPCA*, 14(3):189–204, 2000.

[29] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 227–238, 2005.

[30] N. Bruno and S. Chaudhuri. Physical Design Refinement: The "Merge-Reduce" Approach. In *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, pages 386–404, 2006.

[31] J. Buchholz, Y. Sandoval, and E. Volk. Increase Energy Efficiency of Data Center Cooling Facilities with Airflow Optimization. In *IEEE International Conference on Systems, Man, and Cybernetics, Manchester, SMC 2013, United Kingdom, October 13-16, 2013*, pages 4439–4443, 2013.

[32] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 146–155, 1997.

[33] S. Chaudhuri and V. R. Narasayya. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 367–378, 1998.

[34] G. L. T. Chetsa, L. Lefèvre, J. Pierson, P. Stolf, and G. D. Costa. Exploiting performance counters to predict and improve energy performance of HPC systems. *Future Generation Comp. Syst.*, 36:287–298, 2014.

[35] D. Comer. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.*, 3(4):440–445, 1978.

[36] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985.*, pages 268–279, 1985.

[37] C. R. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life.* 1859.

[38] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *Proceedings of the 2010 International Symposium on Low Power Electronics and Design, 2010, Austin, Texas, USA, August 18-20, 2010*, pages 189–194, 2010.

[39] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254, 2013.

[40] P. J. Drongowski. Basic Performance Measurements for AMD Athlon 64, AMD Opteron and AMD Phenom Processors. 2008. [Online] `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Basic_Performance_Measurements.pdf`.

[41] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[42] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[43] G. F. Franklin, D. J. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2001.

[44] S. Götz et al. Energy-Efficient Databases Using Sweet Spot Frequencies. In *UCC*, 2014.

[45] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[46] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[47] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings*, pages 371–381, 2010.

[48] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[49] D. Hackenberg et al. Introducing FIRESTARTER: A Processor Stress Test Utility. In *IGCC*, 2013.

[50] D. Hackenberg et al. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *IPDPS*, 2015.

[51] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013*, pages 194–204, 2013.

[52] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO '09 - Proceedigns of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, New York, New York, USA, 2009.

[53] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[54] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[55] R. A. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB*, pages 417–428, 2003.

[56] C. Hänsch, T. Kissinger, D. Habich, and W. Lehner. Plan Operator Specialization using Reflective Compiler Techniques. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 363–382, 2015.

[57] S. Harizopoulos et al. Energy Efficiency: The New Holy Grail of Data Management Systems Research. In *CIDR*, 2009.

[58] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 487–498, 2006.

[59] H. Hoffmann. Racing and pacing to idle: an evaluation of heuristics for energy-aware resource allocation. In *Proceedings of the Workshop on Power-Aware Computing and Systems, HotPower 2013, Farmington, Pennsylvania, USA, November 3-6, 2013*, pages 13:1–13:5, 2013.

[60] Hypertransport Technology Consortium. *HyperTransport I/O Link Specification*, revision 3.10c edition, 2010.

[61] IBM. Db2: Column-organized tables. [Online] `http://www.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0060592.html`.

[62] IBM Software Group Information Management. Telecommunication Application Transaction Processing (TATP) Benchmark Description. 2009. [Online] `http://tatpbenchmark.sourceforge.net/TATP_Description.pdf`.

[63] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[64] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 297–308, 2009.

[65] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):585–597, 2011.

[66] Intel. *An Introduction to the Intel QuickPath Interconnect*, 2009.

[67] Intel. Intel Xeon Processor E7 Family Uncore Performance Monitoring Programming Guide. Technical Report April, Intel Corporation, 2011.

[68] Intel. Intel 64 and IA-32 Architectures Software Developer's Guide. 3B(2):31 – 39, 2016. [Online] `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf`.

[69] R. Januszewski, N. Meyer, and J. Nowicka. Evaluation of the Impact of Direct Warm-Water Cooling of the HPC Servers on the Data Center Ecosystem. In *Supercomputing - 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*, pages 382–393, 2014.

[70] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT 2009, 12th International Conference on Extending Database Technology, Saint Petersburg, Russia, March 24-26, 2009, Proceedings*, pages 24–35, 2009.

[71] R. T. Kaushik, T. F. Abdelzaher, R. Egashira, and K. Nahrstedt. Predictive data and energy management in GreenHDFS. In *2011 International Green Computing Conference and Workshops, IGCC 2012, Orlando, FL, USA, July 25-28, 2011*, pages 1–9, 2011.

[72] R. T. Kaushik, M. A. Bhandarkar, and K. Nahrstedt. Evaluation and Analysis of GreenHDFS: A Self-Adaptive, Energy-Conserving Variant of the Hadoop Distributed File System. In *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, pages 274–287, 2010.

[73] A. Kemper and T. Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[74] M. L. Kersten and S. Manegold. Cracking the Database Store. In *CIDR*, pages 213–224, 2005.

[75] T. Kiefer. *Allocation Strategies for Data-Oriented Architectures*. PhD thesis, Dresden University of Technology, 2016.

[76] T. Kiefer, B. Schlegel, and W. Lehner. Experimental Evaluation of NUMA Effects on Database Management Systems. In *BTW*, 2013.

[77] D. Kim et al. Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory). *IEEE Transactions on Computers*, 2013.

[78] T. Kissinger et al. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workloads. In *ADMS*, 2014.

[79] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. KISS-Tree: Smart Latch-Free In-Memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, pages 16–23, 2012.

[80] O. Kocberber et al. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*, 2013.

[81] M. Korkmaz et al. Towards Dynamic Green-Sizing for Database Servers. In *ADMS*, 2015.

[82] O. Kowalke. Boost Coroutines. 2009. [Online] `http://www.boost.org/doc/libs/1_58_0/libs/coroutine/doc/html/index.html`.

[83] W. Lang et al. Rethinking Query Processing for Energy Efficiency: Slowing Down to Win the Race. *IEEE Data Eng. Bull.*, 2011.

[84] W. Lang et al. Towards Energy-Efficient Database Cluster Design. *VLDB*, 2012.

[85] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. *PVLDB*, 3(1):129–139, 2010.

[86] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Vandyke, and C. Vaughan. Energy Delay Product. In *Energy-Efficient High Performance Computing*, pages 51–55. 2013.

[87] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[88] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49, 2013.

[89] J. J. Levandoski, D. B. Lomet, and S. Sengupta. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *PVLDB*, 6(10), 2013.

[90] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of Hadoop clusters. *Operating Systems Review*, 44(1):61–65, 2010.

[91] D. Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. [Online] `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

[92] H. Li, G. Zhu, C. Cui, H. Tang, Y. Dou, and C. He. Energy-efficient migration and consolidation algorithm of virtual machines in data centers for cloud computing. *Computing*, 98(3):303–317, 2016.

[93] D. Lo et al. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *ACM/IEEE*, 2014.

[94] I. Loi and L. Benini. An Efficient Distributed Memory Interface for Many-core Platform with 3D Stacked DRAM. DATE, 2010.

[95] P. J. Magistretti, L. Pellerin, D. L. Rothman, and R. G. Shulman. Energy on demand. *Science*, 283(5401):496–497, 1999.

[96] C. Mohan, H. Pirahesh, and R. A. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992.*, pages 124–133, 1992.

[97] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN 2011, Athens, Greece, June 13, 2011*, pages 17–26, 2011.

[98] S. G. Narendra and A. P. Chandrakasan. *Leakage in Nanometer CMOS Technologies*. Springer US, 2006.

[99] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[100] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689, 2015.

[101] P. O'Neil, B. O'Neil, and X. Chen. Star Schema Benchmark. 2009. [Online] `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[102] Oracle. Oracle SPARC M6-32 Server. [Online] `https://www.oracle.com/servers/sparc/m6-32/index.html`.

[103] E. Pakbaznia and M. Pedram. Minimizing Data Center Cooling and Server Power Costs. In *Proceedings of the 2009 International Symposium on Low Power Electronics and Design, 2009, San Fancisco, CA, USA, August 19-21, 2009*, pages 145–150, 2009.

[104] I. Pandis et al. Data-Oriented Transaction Execution. *PVLDB*, 2010.

[105] M. Poess et al. Energy Cost, the Key Challenge of Today's Data Centers: A Power Consumption Analysis of TPC-C Results. *VLDB*, 2008.

[106] D. Porobic, E. Liarou, P. Tözün, and A. Ailamaki. ATraPos: Adaptive Transaction Processing on Hardware Islands. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 688–699, 2014.

[107] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on Hardware Islands. *PVLDB*, 5(11):1447–1458, 2012.

[108] D. R. K. Ports and K. Grittner. Serializable Snapshot Isolation in PostgreSQL. *PVLDB*, 5(12):1850–1861, 2012.

[109] I. Psaroudakis et al. Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries. In *DaMoN*, 2014.

[110] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *PVLDB*, 8(12):1442–1453, 2015.

[111] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB*, 10(2):37–48, 2016.

[112] F. Raab. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* 1993.

[113] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro Adaptivity in Vectorwise. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1231–1242, 2013.

[114] R. Ramamurthy, D. J. DeWitt, and Q. Su. A Case for Fractured Mirrors. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 430–441, 2002.

[115] J. Rao and K. A. Ross. Making b^+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 475–486, 2000.

[116] J. Rao and K. A. Ross. Making B+-Trees Cache Conscious in Main Memory. *SIGMOD Rec.*, 29, 2000.

[117] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[118] D. Schall et al. Energy-proportional query execution using a cluster of wimpy nodes. In *DaMoN*, 2013.

[119] D. Schall and T. Härder. WattDB - A Journey towards Energy Efficiency. *Datenbank-Spektrum*, 14(3):183–198, 2014.

[120] D. Schall, V. Höfner, and M. Kern. Towards an Enhanced Benchmark Advocating Energy-Efficient Systems. In *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers*, pages 31–45, 2011.

[121] D. Schall, V. Hudlet, and T. Härder. Enhancing Energy Efficiency of Database Applications using SSDs. In *Canadian Conference on Computer Science & Software Engineering, C3S2E 2010, Montreal, Quebec, Canada, May 19-20, 2010, Proceedings*, pages 1–9, 2010.

[122] E. Schrödinger. *Was ist Leben? – Die lebende Zelle mit den Augen des Physikers betrachtet.* Piper Taschenbuch, 1989. ISBN 978-3492211345.

[123] B. Serebrin and D. Hecht. Virtualizing Performance Counters. In *Euro-Par 2011: Parallel Processing Workshops - CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29 - September 2, 2011, Revised Selected Papers, Part I*, pages 223–233, 2011.

[124] SGI. SGI UV. [Online] `https://www.sgi.com/products/servers/uv/`.

[125] Technical advances in the sgi uv architecture. white paper, SGI, 2012.

[126] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner. United States Data Center Energy Usage Report. 06/2016 2016.

[127] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 387–402, 2016.

[128] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160, 2007.

[129] ThinkTank Energy Products Inc. Watts up? Specifications. 2016. [Online] `https://www.wattsupmeters.com/secure/products.php?pn=0&wai=426&spec=5`.

[130] P. Tözün, B. Gold, and A. Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *Proceedings of the Ninth International Workshop on Data Management on New Hardware, DaMoN 1013, New York, NY, USA, June 24, 2013*, page 8, 2013.

[131] Transaction Processing Performance Council. Tpc-energy specification. [Online] `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-energy_v1.5.0.pdf`.

[132] Transaction Processing Performance Council (TPC). TPC Benchmark H. 2014. [Online] `http://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v2.17.1.pdf`.

[133] J. Treibig, G. Hager, and G. Wellein. LIKWID: Lightweight Performance Tools. *CoRR*, abs/1104.4874, 2011.

[134] D. Tsirogiannis et al. Analyzing the energy efficiency of a database server. In *SIGMOD*, 2010.

[135] Y. Tu et al. A System for Energy-Efficient Data Management. *SIGMOD Record*, 2014.

[136] M. Uddin and A. A. Rahman. Server Consolidation: An Approach to make Data Centers Energy Efficient and Green. *CoRR*, abs/1010.5037, 2010.

[137] UEFI Forum. Advanced Configuration and Power Interface Specification. 2016. [Online] `http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf`.

[138] A. Ungethüm et al. Energy Elasticity on Heterogeneous Hardware using Adaptive Resource Reconfiguration LIVE. In *SIGMOD*, 2016.

[139] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, pages 101–110, 2000.

[140] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[141] T. Willhalm, R. Dementiev, and P. Fay. Intel Performance Counter Monitor - A better way to measure CPU utilization. 2012. [Online] `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`.

[142] L. Wu et al. Q100: The Architecture and Design of a Database Processing Unit. In *ASPLOS*, 2014.

[143] Z. Xu et al. Exploring Power-Performance Tradeoffs in Database Systems. In *ICDE*, 2010.

[144] Z. Xu et al. PET: Reducing Database Energy Cost via Query Optimization. *VLDB*, 2012.

[145] Z. Xu et al. Power-Aware Throughput Control for Database Management Systems. In *ICAC*, 2013.

[146] R. Zamani and A. Afsahi. A study of hardware performance monitoring counter selection in power modeling of computing systems. In *2012 International Green Computing Conference, IGCC 2012, San Jose, CA, USA, June 4-8, 2012*, pages 1–10, 2012.

[147] ZES ZIMMER. LMG450 Datasheet. [Online] `http://www.zes.com/de/content/download/689/6677/file/Sensors_LMG600_.pdf`.

List of Figures